NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FILETYPE IDENTIFICATION USING LONG, SUMMARIZED N-GRAMS

by

Ryan C. Mayer

March 2011

Thesis Advisor:	Simson Garfinkel
Second Reader:	Craig Martell

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 10–3–2011	**2. REPORT TYPE** Master’s Thesis	**3. DATES COVERED** *(From — To)* 2008-12-23—2011-03-01

4. TITLE AND SUBTITLE

Filetype Identification Using Long, Summarized N-Grams

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Ryan C. Mayer

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Navy

10. SPONSOR/MONITOR’S ACRONYM(S)

11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A

14. ABSTRACT

Past research into file type identification has employed many different techniques in an attempt to accurately classify files and file fragments including N-gram analysis. However, naïve application of n-grams breaks down when handling n-grams that are greater than two bytes, due to the sparseness of the feature. As a result, other researchers have generally ignored long n-grams for filetype identification. This thesis explores the use of long n-grams for whole file and file fragment classification by building feature distributions of commonly occurring n-grams for single filetypes and using those distributions to classify unknown files and file fragments. This thesis also utilizes summarized n-grams in order to “collapse” similar n-grams within a file type into common n-grams. The algorithms developed to both generate and compare unknown files are presented as well as results from an experiment that was conducted using another researcher’s data set.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	UU	105	

THIS PAGE INTENTIONALLY LEFT BLANK

FILETYPE IDENTIFICATION USING LONG, SUMMARIZED N-GRAMS

Ryan C. Mayer
Lieutenant, United States Navy
B.S., Illinois Institute of Technology, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2011

Author: Ryan C. Mayer

Approved by: Simson Garfinkel
Thesis Advisor

Craig Martell
Second Reader

Dr. Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Past research into file type identification has employed many different techniques in an attempt to accurately classify files and file fragments including N-gram analysis. However, naïve application of n-grams breaks down when handling n-grams that are greater than two bytes, due to the sparseness of the feature. As a result, other researchers have generally ignored long n-grams for filetype identification. This thesis explores the use of long n-grams for whole file and file fragment classification by building feature distributions of commonly occurring n-grams for single filetypes and using those distributions to classify unknown files and file fragments. This thesis also utilizes summarized n-grams in order to "collapse" similar n-grams within a file type into common n-grams. The algorithms developed to both generate and compare unknown files are presented as well as results from an experiment that was conducted using another researcher's data set.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

I must acknowledge God and his wondrous plan he set in motion for me from the beginning of time. His continuous presence always drives me to be the best husband, father, son, and friend that I can be. It's through him I excel and strive to live up to the potential that I have been given. AMDG.

My devoted wife JoAnna. Your continued support, dedication, and sacrifices made it as easy as possible on me to accomplish what I did while attending NPS. It takes a very special woman to be a Navy wife and I am extremely grateful to be married to such an exceptional woman.

Jim Manley, your continued efforts to maintain DOMEX's uptime while conducting my research was much appreciated.

Stefan Axelsson, your correspondence and assistance during my research was very helpful. Your experimental datasets were the center-point of the experiments conducted for my research.

Professor Craig Martell, thank you for agreeing to be the second signer on this thesis. I very much appreciate your involvement and assistance on this thesis and throughout my time at NPS.

Finally, I am very grateful to Professor Simson Garfinkel for agreeing to be my thesis advisor. Your expert knowledge enabled me to produce a thorough research product that I am proud to share with the world. I am extremely thankful for your "steady strain" and active involvement throughout the research process.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

1.1 Background

Filetype identification describes a process of determining the “type” of a file. By “type” we typically mean a name that describes the file’s content (in the case of an ASCII or HTML file); the data structure associated with the file (in the case of JPEG images); or the name of the application program that created the file (in the case of a Microsoft Word document file). Filetypes can be arranged in a hierarchy (Figure 1.1); good filetype identification should produce the type of the highest level of abstraction. For example, a webpage should be identified as HTML, not ASCII. File fragment type identification applies this process to fragments taken from within files.

A file’s “type” is often described by its association, through its file extension, to a particular application that handles files with that particular extension. However, the description of the file’s contents based solely on a file’s extension does not accurately describe what data is contained within the file itself. Primitive data is the most basic form data can be in. These forms can be textual, binary, and random. Compression and encryption adds to the complexity of file type identification since a compressed or encrypted object will have distinct patterns for which file type identification may be impossible without decompression or decryption. A file can consist of only one type of primitive data or be built using blocks of different types of primitive data and arranged in a particular manner that is distinct to the filetype. In the scope of this thesis, the descriptions of filetype for the files used in the learning and test sets take into account the primitive data type(s) present within a file, the manner in which the primitive data types are arranged, and the associated applications that are used to read and write to the files.

Filetype identification can be used to verify that a filetype matches its file extension; to identify the type of a byte string that lacks an extension; or to identify the type of file that a fragment may have originated from. For example, a forensic analyst may search for all files on a computer that are PDF files because they may hold some evidence pertinent to an ongoing case. Searching for files based on the “.pdf” file extension does not guarantee that these files are Adobe Acrobat files, nor does it guarantee that there are not other files that are PDFs but lack the extension. Additionally, when an analyst recovers file fragments from a hard drive, it is useful to be able to determine from what types of files those fragments originate in order to get an idea of what content may have been stored previously on the drive.

To date, the research community's main approach to file type identification has been to use statistical models based on character frequencies and bigrams to predict filetype. This thesis shows that performance is dramatically improved by using longer n-grams common to multiple training files as predictors of filetype.

An n-gram is a string of bytes of *n* length. N-gram usage within the scope of filetype identification is not a new approach. However, previous approaches only utilized short n-grams: n-grams of one to two bytes in length. The long n-gram approach tries to detect strings of n-grams that occur in multiple exemplars of a specific filetype. These long n-grams are then used as features in a machine learning classifier that differentiates files and file fragments of different types.

Detecting the distinguishing characteristics of a filetype is a difficult task. Individual bytes within a file can represent any type of information: text, numerical values coded in decimal or binary, piece of compressed data, piece of encrypted data, etc. These individual bytes are formed into blocks of data, which are in turn combined to form files. Although the meaning of the bytes can be readily inferred when presented to the appropriate program, the forensic investigator may not have the correct program. Also, some byte arrangements may have ambiguous codings. Being able to understand these codings is important to enable automated forensic processing.

One way to identify filetype is to identify distinguishing characteristics. These characteristics can be used as features in a traditional machine learning classifier. There has been more than a decade of research on this problem. It is complicated by many factors. Some characteristics may uniquely identify a single filetype, while others may identify several possible filetypes or rule others out. Some characteristics may only be relevant when present at certain locations within a file. For example, the string JFIF identifies a JPEG, but only when present near the beginning of a file. This makes fragment identification harder as the location of the fragment in the file is generally unknown. A danger when using machine learning algorithms to identify file type is that the algorithms may be trained on a feature that is not indicative of filetype. For example, a system that is trained to distinguish JPEG files from HTML files may actually be inadvertently be trained not on the HTML vocabulary, but on the frequency of n-grams in the HTML files' English text. Such a system might be unable to recognize HTML files containing Arabic text.

1.1.1 Filetype Identification of Complete Files

There are two closely related yet different problems with filetype identification: identifying the type of complete files and identifying the type of file fragments. Whole file identification is relatively straight forward for known filetypes as long as the file is intact. Two approaches to whole file identification involve searching for file header/footer information, and attempting to decode or interpret the file's internal structures. Techniques utilizing this approach use libraries and signature databases.

Simple file formats are formats that contain a single type of primitive data. Text files, compressed files, and encrypted files are examples of simple file formats. The structuring within these simple formats will be characteristic of the primitive data that is contained within. For textual files, individual byte values will cluster around the values of printable ASCII characters and the distribution of n-grams will reflect the characteristics of the written language. For compressed and encrypted file formats, individual byte values will be evenly distributed and there will be no common n-grams. However, the header and footer information necessary to perform the decompression or decryption of these files will be present in these files and can be used as a signature for filetype identification.

While both compressed and encrypted data are indeed primitive data, there still remains the question of what data is contained within the compressed or encrypted data. If the algorithm used to compress or encrypt the data can be determined, then it is frequently possible to uncompress or decrypt the data and perform further identification.

Complicating matters further still, many common filetypes are container files. Container files are files that have the ability to encapsulate different types of data within them. Examples of container files are Microsoft Word Document files, ZIP files, and Adobe Acrobat files. For example, PDF files can directly embed files of a different filetype such as JPEG files. Container files have fields that mark the locations of encapsulated data and appear as internal structures that are distinct to the filetype. Whole file identification can use these structures within a container file as distinct characteristics of its filetype.

1.1.2 File Fragment Identification

File fragment identification is significantly more complex. With whole files, header/footer information and any internal structures that a file may contain are intact. However, a file fragment can originate from any location in a file: header/footer and internal structures may not be present in the fragment, making identification difficult. File fragment identification has so far relied on the following general ideas: statistical distributions of bytes, recognizing internal structures of file types, entropy measurements, and n-gram analysis. Whole file identification projects have had successes at being able to accurately identify filetype. However, file fragment identification has had limited success.

The hardest part about file fragment identification is the fact that, for a given a file fragment, there is no way to tell the offset in the original file from which it came from. Many filetypes can be identified by the structure of the whole file without having to rely solely on header/footer information. JPEG and MP3 files are such examples, as both have structuring data distributed throughout the file (segments in a JPEG and frames in an MP3) that can be predictors of the filetype when the header/footer information is not present. However, file fragments can be considerably smaller in size (i.e.,: a 512-byte sector from a hard drive) than the original file. Bytes within the fragment may lack the structuring information, making their type ambiguous. Without any reference as to where the fragment originated, the fragment's type may not be identifiable.

The majority of file fragment approaches rely on characteristics of a whole file to be present within the fragment itself. For instance, a file fragment that originated from a compressed file should have the same characteristic high data entropy. However, there are a multitude of filetypes that employ compression within the format to enable efficient storage. At a fragment level, these filetypes will be difficult to differentiate because high entropy data will have relatively even distributions of bytes that can look the same although they are of different filetypes.

Another difficulty in to the matching of fragments to filetypes are file containers that can contain multiple primitive types. In this case, it would be impossible to determine if a fragment is the primitive type or a primitive that has been embedded in a container. This causes many classification errors.

1.1.3 Internal Structuring of Files

At the highest level, all files are a collection of bytes. In order for an application to be able to read and write to a particular file of a filetype, it needs to be designed to properly handle the properties of the targeted filetype that are contained in the encoding of a file header/footer and a file's metadata. These properties give filetypes the ability to structure their data in a standard way and provide for logical access to a file's content. Structuring has to be common among files of a given filetype in order to allow applications access to the data from the files. Therefore, the structuring of a filetype can be used as distinctive characteristics for filetype prediction.

Figure 1.1 shows some exemplar filetypes arranged within a tree structure. Under the root node, the figure lists five exemplar categories of filetypes. These categories all package their relative data in a specific way. For example, chunk based files use markers to signal the beginning of a new chunk of data, directory based files use a directory system to store data, encapsulated filesystem filetypes mirror the logical layouts of common filesystems, and stream based filetypes (such as the ones listed in Figure 1.1) use packets to store a data-stream for playback of audio/visual content. For each of the examples given, one could use the structuring of the filetypes for filetype identification since the structuring provides distinctive byte patterns or n-grams. In turn, if an unknown file is discovered to contain a distinctive n-gram associated with a known filetype, then the filetype of the unknown file can be determined with some degree of confidence.

While chunk based, directory based, encapsulated filesystem based, and stream based provide important structuring data, one filetype from Figure 1.1 was left out of the previous paragraph's discussion: textual based files. Textual based files, like the ones shown in Figure 1.1, have extremely primitive structuring and in some cases none at all (TXT files). Filetypes such as comma separated value (CSV) or true-type fonts (TTF) have a structure. However, the structure consists of a single ASCII value that occurs with a much greater frequency than any other ASCII character in the file (the ASCII comma value for CSV files and the ASCII "tab" value for TTF). While these files have a structure associated with them, single byte values cannot provide enough of a distinctive n-gram. Textual based files (as will be presented within this thesis) make precise automatic filetype identification difficult using the long n-gram approach due to the lack of structuring and risk making filetype predictions based on a file's content rather than the structure of the file.

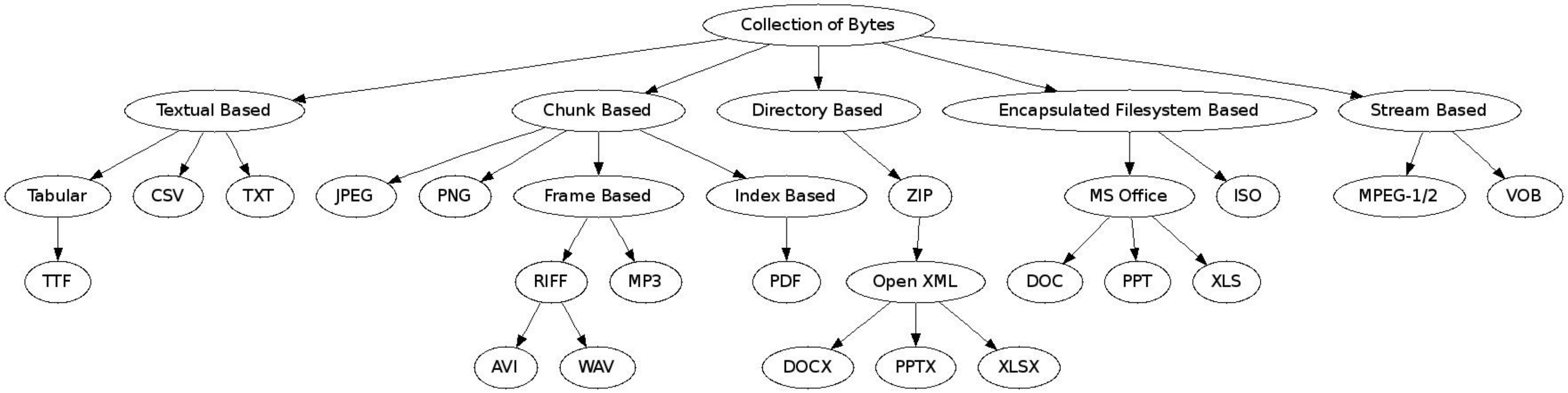

Figure 1.1: File hierarchy of some exemplar filetypes

1.2 Motivation

The goal of this research is to determine if long n-grams are useful predictors of a file or file fragment's filetype. Previous research limiting the size of the n-grams to lengths of one to two bytes could very well miss n-grams of greater lengths that are indicative to a specific filetype.

Primitive data may have distinct characteristics associated with it. These characteristics may come in the form of long n-grams in which case the presence of these long n-grams in primitive data would indicate the file or file fragment's filetype. However, techniques for analyzing n-gram characteristics within files have been limited to date to short n-grams of one to two characters in length. This appears to be the result of a common implementation approach, not the underlying science.

Previous researchers have used n-dimensional arrays to hold all counts of the occurrences of the n-grams in a file and keep track of the n-grams that did not appear. These methods easily handle n-grams of length two since there are only 65,536 possible 2-grams (256×256). However, with each additional byte added to the n-gram's length, the size of the array required to store all counts of n-grams increases exponentially. This exponential growth creates unsatisfiable memory constraints. It also breaks classical machine learning algorithms as the data contained within such arrays become too sparse to classify. Therefore, these technical approaches do not scale well for long n-grams.

The approach presented within this thesis searches multiple examples of a filetype for commonly occurring long n-grams and then uses these long n-grams as predictive features. That is, the long n-grams are used to build a dictionary that is specific to and characteristic of each filetype. Since the presence of these features is used to identify a file, there is no need to keep track of n-grams that did not occur within the file (as other researchers have done). A distribution of the distinct characteristics or features of a filetype can be generated through sampling sets of files.

Throughout the course of this thesis, the term *long n-gram* will refer to n-grams with sizes greater than 2-bytes. The frequency of occurrences of a long n-gram within a single file is not important: what is necessary is that the long n-gram is present in a majority (or all) of the files of a single filetype. Likewise, there is no requirement that all long predictive n-grams be of the same length. In fact, they are not.

The approach developed here is based on an observation as to how humans perform manual filetype identification. In our experience, humans do this by looking for sequences of characters that they recognize from having seen many other exemplar files of a given type. For example, Figure 1.2 shows the fist 1024 bytes of the PDF file of a previous version of this thesis. Casual inspection of this figure shows that some of the strings are readily recognizable as conferring *type*, while others are clearly aligned with the document's *contents*. For example, the string `%PDF-1.4` is a header readily associated with PDF files, while `/Length`, `/FlateDecode`, `endstream` and `endobject` are all associated with the PDF vocabulary. On the other hand, `/nps_logo_3clr_cymk.pdf` is the file name of the NPS logo that was included in the thesis document and is indicative of the file's content, not its type.

1.3 Outline of This Thesis

Chapter 2 of this thesis presents previous research into filetype identification of both complete files and file fragments in order to give the reader a solid background on filetype identification research to date. Chapter 2 also presents various implementation that have been built on the research presented in order to make the reader aware of the various tools available to preform filetype identification.

Chapter 3 presents the concept and intuition of using long n-grams for filetype identification. This chapter includes multiple figures in order to help illustrate how the long n-gram approach is useful.

Chapter 4 presents experiments conducted using the long n-gram appraoch with software develped for this thesis research.

Appendices contain lists of the files and file fragments utilized for the experiments presented in Chapter 4.

```
0000    25 50 44 46 2d 31 2e 34 0a  25 d0 d4 c5 d8 0a 34   %PDF-1.4.%.....4
0010    20 30 20 6f 62 6a 20 3c 3c  0a 2f 4c 65 6e 67 74    0 obj <<./Lengt
0020    68 20 35 39 36 20 20 20 20  20 20 20 0a 2f 46 69   h 596       ./Fi
0030    6c 74 65 72 20 2f 46 6c 61  74 65 44 65 63 6f 64   lter /FlateDecod
0040    65 0a 3e 3e 0a 73 74 72 65  61 6d 0a 78 da a5 54   e.>>.stream.x..T
0050    cb 6e db 30 10 bc fb 2b 78  94 80 8a e1 f2 25 b2   .n.0...+x.....%.
0060    3d a9 b6 ec a8 b0 e5 d4 52  0a 04 69 0e 7e 28 a9   =.......R..i.˜(.
0070    50 bf 2a db 29 fc f7 5d 89  72 6a a7 45 73 28 04   P.*.)..].rj.Es(.
0080    88 14 77 38 c3 9d 5d 8a 91  27 c2 c8 a0 c3 de 18   ..w8..]..’......
0090    01 df 8c 00 01 ab 88 12 82  86 96 93 f9 aa f3 e3   ................
00a0    25 50 bf ff b6 70 95 ac 80  f4 36 9d cf f8 9c 42   %P...p....6....B
00b0    41 cd 12 9c d1 7c cc 3b 57  7d 09 84 4b 1a 9a 10   A....|.;W}..K...
00c0    48 fe 48 b8 56 54 2b 4b 14  13 d4 32 49 f2 05 b9   H.H.VT+K...2I...
00d0    f7 52 9f 33 2f f2 41 2a ef  8b 0f 42 79 d1 d0 7f   .R.3/.A*...By...
00e0    c8 3f 91 40 03 65 1c 39 b9  a5 c6 18 07 be 19 67   .?.@.e.9.......g
00f0    f9 60 12 f5 6e 7d 5d 6f b2  ca cb e3 06 2d 0d 15   .`..n}]o.....-..
0100    e2 15 38 eb 5e 8f c7 0d 59  73 10 90 54 48 cd eb   ..8.^...Ys..TH..
0110    83 04 02 e1 1a 0f 2d 81 1a  29 1c 7c 34 4e f3 78   ......-..).|4N.x
0120    12 df f9 96 7b ef fc 80 2b  e6 75 a3 61 d2 1f 4f   ....{...+.u.a..O
0130    d2 24 7a 61 39 4b 47 6a aa  01 35 71 e4 42 3b 92   .$za9KGj..5q.B;.
0140    fc 3a ce 92 ec b7 26 50 ab  94 d3 34 96 82 42 78   .:....&P...4..Bx
0150    c8 28 53 a1 83 f7 93 61 ec  a4 f2 bb 9b 76 96 f4   .(S....a.....v..
0160    e2 34 4f fa 49 d7 e5 97 8c  53 b7 7e 9b 25 e9 c0   .4O.I....S.˜.%..
0170    4d d3 00 4d 18 9d 64 e4 b9  0c f0 10 e7 12 9d d0   M..M..d.........
0180    54 32 70 32 b3 a3 b3 54 30  1a 02 bf 8c 4d 8e d3   T2p2...T0....M..
0190    75 9b 2d 75 e3 68 7a 2c aa  66 43 9d 6c e3 ea 19   u.-u.hz,.fC.l...
01a0    9e 33 26 1d 1b 48 4e b9 0e  31 2c d0 80 36 9c 7f   .3&..HN..1,..6..
01b0    2b 76 e5 ce 11 45 8b e7 72  b7 a9 de fb 01 30 a3   +v...E..r.....0.
01c0    c1 eb 55 be 52 1e aa 08 60  5e 56 ae 76 9b 56 79   ..U.R...`^V.v.Vy
01d0    30 ad be 32 c6 d7 df 7d 0c  14 cb 86 9e d5 02 54   0..2...}.......T
01e0    ca d6 d8 ac 98 6f d6 0b 07  9f 14 d3 45 e1 68 2d   .....o......E.h-
01f0    98 4b da 6e 35 2d 9f 4e 79  54 fb 62 d9 b0 75 e2   .K.n5-.NyT.b..u.
0200    fc ac 8d 01 2d e3 36 a4 12  c2 ba 53 ef 1f 18 59   ....-.6....S...Y
0210    60 00 35 a9 b0 86 fc 6c 60  2b 22 b8 c4 71 49 32   `.5....l`+"..qI2
0220    6c f3 8b cd 14 ac 25 80 fd  89 bd fc 8f fd 88 16   l.....%.........
0230    8a 1a ae ff e0 90 3a a4 06  0d fd 1f 8e 3a 89 b7   ......:......:..
0240    f7 9f e7 70 ba 91 c0 f0 f6  d9 a6 85 c1 70 0a 02   ...p.........p..
0250    89 b0 94 ca b4 37 32 f2 b9  f2 b6 db ca 47 6f 37   .....72......Go7
0260    75 45 9e 9b b2 b4 e6 3f d6  d1 4d e5 3e b6 87 d9   uE.....?..M.>...
0270    b2 9c bb 79 83 2f 96 c5 74  57 7c 70 2b 8b 72 b7   ...y./..tW|p+.r.
0280    af ca 59 7d c1 0f fb f2 54  ec 53 77 1c d6 cb 72   ..Y}....T.Sw...r
0290    55 ee 91 b7 2e d0 eb 9f 13  16 ec 17 e1 4d 15 3f   U............M.?
02a0    0a 65 6e 64 73 74 72 65 61  6d 0a 65 6e 64 6f 62   .endstream.endob
02b0    6a 0a 33 20 30 20 6f 62 6a  20 3c 3c 0a 2f 54 79   j.3 0 obj <<./Ty
02c0    70 65 20 2f 50 61 67 65 0a  2f 43 6f 6e 74 65 6e   pe /Page./Conten
02d0    74 73 20 34 20 30 20 52 0a  2f 52 65 73 6f 75 72   ts 4 0 R./Resour
02e0    63 65 73 20 32 20 30 20 52  0a 2f 4d 65 64 69 61   ces 2 0 R./Media
02f0    42 6f 78 20 5b 30 20 30 20  36 31 32 20 37 39 32   Box [0 0 612 792
0300    5d 0a 2f 50 61 72 65 6e 74  20 37 20 30 20 52 0a   ]./Parent 7 0 R.
0310    3e 3e 20 65 6e 64 6f 62 6a  0a 31 20 30 20 6f 62   >> endobj.1 0 ob
0320    6a 20 3c 3c 0a 2f 54 79 70  65 20 2f 58 4f 62 6a   j <<./Type /XObj
0330    65 63 74 0a 2f 53 75 62 74  79 70 65 20 2f 46 6f   ect./Subtype /Fo
0340    72 6d 0a 2f 46 6f 72 6d 54  79 70 65 20 31 0a 2f   rm./FormType 1./
0350    50 54 45 58 2e 46 69 6c 65  4e 61 6d 65 20 28 2e   PTEX.FileName (.
0360    2f 6e 70 73 5f 6c 6f 67 6f  5f 33 63 6c 72 5f 63   /nps_logo_3clr_c
0370    79 6d 6b 2e 70 64 66 29 0a  2f 50 54 45 58 2e 50   ymk.pdf)./PTEX.P
0380    61 67 65 4e 75 6d 62 65 72  20 31 0a 2f 50 54 45   ageNumber 1./PTE
0390    58 2e 49 6e 66 6f 44 69 63  74 20 38 20 30 20 52   X.InfoDict 8 0 R
03a0    0a 2f 42 42 6f 78 20 5b 30  20 30 20 32 32 32 20   ./BBox [0 0 222 
03b0    31 35 33 5d 0a 2f 52 65 73  6f 75 72 63 65 73 20   153]./Resources 
03c0    3c 3c 0a 2f 50 72 6f 63 53  65 74 20 5b 20 2f 50   <<./ProcSet [ /P
03d0    44 46 20 5d 0a 2f 45 78 74  47 53 74 61 74 65 20   DF ]./ExtGState 
03e0    3c 3c 0a 2f 47 73 33 20 39  20 30 20 52 0a 2f 47   <<./Gs3 9 0 R./G
03f0    73 31 20 31 30 20 30 20 52  0a 2f 47 73 32 20 31   s1 10 0 R./Gs2 1
```

Figure 1.2: A hex view of the first 1024 bytes of a previous version of this thesis

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Prior Work

2.1 Techniques for Filetype Identification

2.1.1 Byte Frequency and N-Gram Analysis

McDaniel introduced the field of statistical filetype identification with his master's thesis, which used byte frequency to classify filetype [1].

In his work, McDaniel proposed an algorithm with three options to generate "file fingerprints." The first option generates byte distributions from all the bytes contained within a file. The distributions are 256 element vectors that hold the counts of each byte value (corresponding to the index in the array) in an entire file. These values are then normalized. Option two utilizes a 2D array (size 256×256) to detect co-occurrences of byte pairs in a file. This is useful for file types such as HTML where "$\boxed{<}$" and "$\boxed{>}$" ASCII values will appear with higher frequencies and in roughly equal proportions and will be more frequent than an ASCII string of "$\boxed{<}\boxed{<}$". This option further helps to differentiate characteristic file types. The third option uses a file's header and tail to generate two arrays (one each for the header and the footer) for byte frequency correlation for filetypes that do not have "strong" byte frequencies.

McDaniel's algorithm generates file fingerprints for known file types by generating a running frequency distribution over all files used as input. This distribution is the so-called *file fingerprint* for a particular type of file. These fingerprints can the be used to compare an unknown file's byte frequency distribution (BFD) to the fingerprint. The algorithm creates *scores*, ranging from 0 to 1, for each file as it is compared to the fingerprint. A *score* is the average of the differences between each byte value from the fingerprint and the unknown file. These scores are then used to classify an unknown file by taking the highest score relating to a fingerprint.

McDaniel's content based file detection suffered low accuracy ratings for options one and two (28% and 46%, respectively). However, option three generated 96% accuracy. McDaniel only applied his method to entire files, although the algorithm could be applied to file fragment classification using options one and two. The third option will not be effective against file fragments since a fragment will rarely contain either the header and tail of a file, and never both.

Li *et al.* introduced a method called *fileprinting* similar to McDaniel's for file type identification of files using byte distributions [2]. However, they proposed using centroid models instead of distance scoring and their normalizing strategy was to normalize byte frequencies by the file length. Their approach also was designed to deal with truncated network traffic. The authors ran their experiments on the first n-bytes of a file (n being 20, 200, 500, and 1000 bytes) as well as the entire file. Li *et al.* experienced very good accuracy rates for their truncated data experiments. Interestingly, their accuracy results for using the entire file contents were significantly lower than only using the first 20 bytes, indicating their algorithm was really just recognizing file headers and was actually confused by file contents. Unfortunately, Li *et al.* did not test their method against file fragments taken from the middle of files.

The Oscar method [3], developed by Karresand, utilizes a centroid byte frequency analysis similar to Li's method that analyzes binary data within disk clusters and RAM pages. Karresand further extended his Oscar method with a new metric called "rate of change" (RoC) which he described as the absolute value of the difference between two consecutive byte values in a data fragment. Mainly implemented to identify JPEG data, the Oscar method was validated on a single concatenated file consisting of 57 different file types taken from a Windows XP SP 2 machine. Karresand describes three experiments that tested the detection rates of the Oscar and RoC methods: JPEG data, Windows executable data, ZIP data. The results for the JPEG accuracy was excellent, although not too surprising since a JPEG file has a distinct and regularly appearing byte pattern of `0xFF` `0x00` throughout its structure. The BFD method produced a 97.4% true positive rate with a false positive rate of 0.005%. The RoC method eliminated the false positive rate and had a better true positive rate of 99.2%. For the Windows executable detection, neither algorithms were able to produce better than a 12% detection rate. However, the false positive rate never exceeded 1.9%. Lastly, the ZIP data detection had good detection rates of 46%–84% but had false positive rates ranging from 11%–37%

2.1.2 Sliding Windows

Hall and Davis explored whether a sliding window could improve filetype identification [4]. This research utilized entropy and compressibility measurements for each sliding window. For each filetype, the measurements were averaged and a standard deviation was calculated for each corresponding data point to obtain a profile plot. The next step was to develop a mechanism to associate a filetype to a file. The first mechanism was called a "point-by-point delta." This approach is a variance method that compares a sliding window calculation to a profile plot of a filetype. At each point throughout a profile, the sliding window value is subtracted from the profile's average value. The absolute value of this subtraction is then accumulated for all points in the profile to form what they called a "goodness of fit." The smaller this value was indicated how closely the file related to a particular profile. Therefore, the best guess as to the file's type was the smallest "goodness" calculation for a particular filetype profile.

The second method of classification used was the Pearson's Rank Order Correlation. This method calculates how well two sets of data correlate. The higher the correlation, the better the two sets of data are said to match. This method was applied with a sliding window by tracking how well the values of points from both the file and the profile rose and fell together. Any small or negative correlation values indicated that the file and the profile plot did not have similar plots.

Hall and Davis utilized two sets of files for their experiment. The first set was a reliably labeled test set that was obtained from the lead researcher's personal computer. This set contained approximately 73,000 files and represented 454 file types identified by the Unix *file* utility. The second data set consisted of over 265,000 files with 109 different filetypes. This set was obtained from ManTech; the experimenters later discovered that this set contained improperly labeled files or files of a size not conducive to sliding windows. For example, the authors reported that of the 100 ZIP files contained in set 2, only 83 were 9000 bytes or larger. This relatively small size will not provide an accurate average profile of ZIP data. Therefore, the authors claim that set 2 was an unreliable set for validating their approach.

Hall and Davis reported some good results for their sliding window approach on set 1. For the entropy measurements, both methods of association worked well. For the point-by-point delta mechanism, the results ranged from 0 to 100% accuracy with 20 out of 25 of the reported classification results were equal to or greater than 80%. The correlation mechanism had accuracy ranging from 12% to 100% with the majority of the accuracy ratings clustering around 50% to 75%. The compressibility measurements did not perform as well as the entropy measurements. Significant amounts of both association methods had zero percent accuracy ratings. However, some filetypes such as DOC or XLS files had better accuracy results over the entropy measurements.

Another sliding window technique developed by Moody and Erbacher [5] utilizes a statistical approach to file type identification. Their technique called, SÁDI (Statistical Analysis for Data type Identification), involves averaging sliding window values across a file and then utilizing applied statistical techniques to analyze and graph the structure of the files. Moody and Erbacher explicitily did not use container files. Their work had the goal of being able to identify unique and consistent patterns within a base file that can be used to differentiate it from other base files. The pitfalls to this approach (as was noted in their evaluation) is the inability to differentiate file structures of closely related data types. This lead the researchers to develop a two-pass technique for analyzing files. During the first pass, the analysis code employs statistical techniques and attempts to identify data blocks of the unknown file to statistical structures of known data types. The second pass then tries to identify unusual patterns within the computed statistics. This two-pass technique applies to the research conducted within the scope of this thesis because the two pass technique could possibly be used to identify variations within a data type and allow for more fine grained identification.

2.1.3 Normalized Compression Distance

Axelsson developed a technique that analyzed the normalized compression distance (NCD) of file fragments in order to classify them [6]. The compression distance is a measure of how distant two data vectors are when compressed concatenated and individually. The better the concatenated vectors compress compared to individually, the more similar the two data vectors are. Normalization is necessary to account for differences in vector length. Axelsson's NCD approach also required a classification technique and for this: k-nearest-neighbor was utilized with different values of k. The k-nearest feature vectors were selected by which ever class (filetype) had the largest number of votes.

Unlike all previous researchers who used private data sets that could not be distributed due to copyright and privacy concerns, Axelsson chose to utilize the freely available file corpus of 1 million U.S. government documents assembled by Garfinkel *et al.* [7]. He chose this corpus because his work can be reproducible by other researchers and the corpus contains many different common filetypes that any forensic researcher would encounter in the field. Axelsson's experiment focused on 512-byte file fragments since this fragment size is the smallest block size available on current magnetic media. The compression algorithm used in this experiment to calculate the compression distance was the bzip2 algorithm because it can comfortably handle the lengths of the file fragments (2 x 512) without much overhead. Axelsson ran his experiment ten times, each using a randomly chosen subset of 3000 blocks from the corpus with values of k for each trial being from 1 to 10. For each trial, ten files for each filetype were chosen at random to generate the classification sets. One drawback to Axelsson's experiment (noted by Axelsson) is that the test data was not verified to be properly labeled. Instead, he relied on a file's extension to accurately represent the data contained within it.

Axelsson stated his results were unremarkable given that the average hit rate for the k values of 1 to 10 ranged from 36% down to 32% respectively. However, he also included average accuracy ratings of each k-value for each type of file classified. File types of *eps, java, csv,* and *ttf* all had accuracies of 63% to 68% averaged over the 10 k-values. Other filetypes such as *jpg, gif, pptx, pps, gz,* and *png* all had very low accuracy ratings for all k-values ranging from 3% to 12%. This is still better than a random classifier, which would have had accuracy ratings of around 3.5% for each filetype.

Although the overall accuracy ratings were relatively low, Axelsson's experiment also produced some interesting patterns in the misclassifications. The classifier misclassified large amounts of files as *docx*. For example, 495 out of 1375 *pps* files, 470 out of 1067 *png* files, and 381 out of 1141 *jpg* files were misclassified as *docx* files. DOCX files are actually ZIP files with specific structured contents [8]. Axelsson did not include a full confusion matrix due to space restrictions in the paper but he did report that most every file appeared to be a *docx* file to the classifier. This result, although not surprising, shows that filetypes containing large zlib-compressed regions tend to classify together using his technique. Also of interest in Axelsson's experiment was that a k-value of *k=1* gave the best classification results over the other values, indicating that the method worked best when it would find a single exemplar file that was close to, in NCD space, the file being tested.

Thus, while the accuracy of the NCD approach implemented by Axelsson did not produce fantastic results, it did produce some accurate results that were much better than random classifications. Further, if his experiment was conducted using a properly labeled test set, perhaps the classifications could have been significantly higher. Close analysis of Axelsson's algorithm indicate that it is actually identifying common n-grams and using them to classify the fragments. His implementation utilized the bzip2 library to compress the file fragments. The bzip2 library is based off the Lempel-Ziv-Markov chain algorithm that uses a dictionary compression scheme to remove repeating data patterns within a set of data. The removed patterns are listed within the dictionary with pointers to where the patterns were removed so that the data can be uncompressed by placing the removed patterns back into the data. The information stored within the dictionary are n-grams that were repeating features in the data stream that could be removed to compress the data. Therefore, by using the compression distance between file fragments, Axelsson was classifying fragments based on the amount of reoccurring n-grams that could be removed from an unknown fragment and compared to the amount of n-g rams removed for a whole exemplar file of a single filetype.

2.1.4 Specialized Approaches

Roussev and Garfinkel [9] argue that header/footer analysis of whole files and statistical analysis and machine learning applied to file fragments are generally unsuccessful because they fail to take into account the inherent internal structures of filetypes. They state that researchers will often pursue a generalized approach to a problem and tend to shy away from "reinventing the wheel." These generalized approaches (statistical analysis and classical machine learning) are usually borrowed from other fields where they have had good results but end up having limited or no success across the entire field of filetype identification.

Many popular filetypes (PDF, JPEG, GZ) in use today have definitive structures within. A specialized approach will take into account these characteristic structures and make a classification based on the presence of these structures in a file or file fragment. Roussev and Garfinkel outline four primary goals that researchers should strive to achieve when developing specialized approaches. The first goal is accuracy and it is critical when dealing with terabyte-scale targets. High classification rates (99% and greater) are necessary for a specialized approach to be effective with a large scale. The second goal is reliable error rate estimates. Every classifier should have error rates that have been established by using publicly available data sets. The third goal is clear results. A classifier should strive to have no false positives and false negatives but in general false positives are worse. It is better for a classifier to report that it is unable to classify rather than "guess" at a classification. Lastly, the fourth goal is line-speed performance. This goal states that a classifier should be able to keep up with a bulk I/O transfer from a secondary storage device.

Roussev and Garfinkel conclude that specialized approaches for specific data formats can be superior to generalized filetype identification as long as a researcher is willing to study file format specifications, view the binary structure of files by hand, and create hand-crafted recognizers. Of course, this approach is also labor intensive

2.1.5 Validation of Internal Structures

Garfinkel *et al.* [10] proposed validating the internal structures of multimedia files as a way to perform filetype identification. They argued that internal structures may be present within a given filetype that can be used to characterize the filetype and be used to differentiate it from other filetypes. Rather than developing a general approach to filetype identification, like byte frequency, Garfinkel *et al.* state that purpose built functions can be developed to recognize specific internal structures of a filetype.

Container files can consist of many different types, a single fragment from a container file can actually contain multiple types. Therefore, the authors adopted the term *filetype discrimination* rather than using filetype identification. This term recognizes that a file fragment taken from the middle of a JPEG file will not have any apparent difference with a fragment taken from the middle of a JPEG file embedded in a PDF file. In this case, without looking at the neighboring fragments (if they are available), this fragment will not be able to be identified as coming from a PDF file. Therefore, while it may not be possible to properly identify the type of file a fragment came from, it is possible to at least classify the type of file the fragment relates to.

Many multimedia files contain repeating frames that have a variable or fixed length offset and may be able to be recognized by specific byte patterns. For example, the *JPEG* file format uses the hexadecimal string `0xFF` `0x00` to indicate the beginning of segments. Each of these segments contain Huffman-coded data that make up the image stored within the file. Garfinkel *et al.* [10] highlighted a JPEG discriminator they built that looked for blocks of data that contained high entropy data and contained more bigrams of `0xFF` `0x00` that would regularly occur in high entropy data. The discriminator would accept a block as a JPEG block if it had a high entropy (HE) measurement of more than n distinct unigrams and had at least a predetermined number of `0xFF` `0x00` bigrams (LN; Least N-gram count). In order to tune the discriminator, various values of HE and LN had to be tested to determine which combination gave the best results. This was done with a grid search utilizing a set of 30 million 4KiB file fragments from the *govdocs1* corpus [7] for each iteration of the tuning. The best measure of HE turned out to be 220 with three values of LN clustering around the highest accuracy ratings: LN = 1 produced 98.91% accuracy, LN = 2 produced 99.28% accuracy, and LN = 3 produced 99.08% accuracy.

Garfinkel *et al.* also discuss two other discriminators: MP3 discriminator with a 99.56% accuracy and a Huffman-coded data discriminator with a 49.5% accuracy for 4KiB blocks and 66.6% accuracy for 16KiB blocks. Although the Huffman-coded discriminator's accuracy is not as impressive as the JPEG or MP3 discriminator, Garfinkel *et al.* report that the discriminator rarely mistook encrypted data for compressed data. This is significant since compressed data and encrypted data will have relatively close entropy and the other algorithms discussed in this chapter had difficulty distinguishing the two types.

2.2 Existing Filetype Identification Implementations

2.2.1 The Unix *file* Command

The Unix *file* command is probably the most widely used filetype identification system today. The file command analyzes file headers and footers to identify filetypes. This utility has been present in every Unix release since the November 1973 Research Version 4. When System V was released in 1983, the *file* utility included a list of so-called *magic* numbers that it utilized to classify files. Magic numbers are invariant bytes of data located at small offsets from the beginning of a file that are specific to a particular filetype. These magic numbers can then be searched for and cross referenced by the external list of magic numbers to determine what the file is.

When executed, the *file* command performs three sets of tests in order to classify the file given as an argument. The first test performed is a filesystem test. This test performs a Unix *stat* call in order to determine if the file is empty or some type of a special file such as a symbolic link or system device (i.e., */dev/tty*). The second test examines the entries in the external magic number listing to see of the file has any data within a small fixed offset of the beginning of the file that corresponds to an entry in the external listing and if so reports the description of the filetype listed in the external listing. If the first two tests fail to identify the file, the third test of determining if the file is a textual file is performed. Recent versions of the *file* utility analyze the content of the file and determine if the ranges and sequences of bytes correspond to various character sets. These character sets include sets such as ASCII, ISO-8859-x, UTF-8, and UTF-16. If the utility can determine the character set, it will then attempt to determine if the file is a programming language file. The program looks for certain keywords that are unique to a specific programming language. For example, the keyword *struct* is specific to the C programming language. If any keywords can be found, the utility will then output the name of the programming language. The third test is reportedly less reliable than the first two and is therefore performed last. If all three tests fail to determine the type of the file, the string “data” is output.

The *file* utility is essentially using a modified long n-gram approach to determine the type of a file. A header/footer of a file can contain specific character n-grams that the utility matches to its library. Here the n-gram approach is modified in that certain n-grams are only accepted if they occur at specific locations within the file. If the file is not a special file and fails the magic number test, the utility is again using n-grams to test for textual content. If content of a file is textual, it again uses long n-grams to determine if the textual data corresponds to the programming languages that it has in its library.

The *file* utility is a quick and straightforward tool to use to determine a file's type. However, there are downfalls to it. First, the *file* utility can be spoofed by opening up a file within a hex editor and changing the magic numbers to misrepresent the file. And second, it does not work on file fragments, with the exception that the fragment contains a descriptive header/footer data or is a fragment from source code.

2.2.2 TrID

TrID [11] is a closed-source but freely available utility that identifies files based on their binary signatures. The program uses a database of definitions that describe the recurring patterns (binary signatures) of all the filetypes in the database and matches a file to the correct filetype. The developer for TrID states on the project's website that the publicly available database currently contains 4038 (as of November 2010) filetype definitions and is always expanding to include new filetypes. TrID users also have the option to define a local database that can be used to store filetype definitions that pertain to filetypes that are unique to a group of people. TrID gives the user the option to append file extensions to files it identifies or show extended information about the file (i.e.,: display ID3 information about an MP3 file).

2.2.3 Oracle Outside in Content Access

Outside In Technology [12] is a suite of software development kits produced by Orace and can be used for document extraction, conversion, and viewing of 500 file formats. The API that handles filetype identification for Outside In is called File ID. Its method of operation is not documented. The Outside In SDK is available for both Windows and Unix platforms.

2.2.4 DROID – Digital Record Object IDentification

Developed by the UK National Archives [13], DROID (Digital Record Object Identification) recognizes over 200 file formats. It was designed to scan repositories of millions of files and identify them based on signature information. DROID was developed for digital preservation and produces reports that are geared at informing users what content they have within their repositories. DROID also will look inside archive files (ZIP, GZ, etc.) and analyze the content contained within those files. This utility is platform independent and written in Java.

2.2.5 GDFR – Global Digital Format Registry

The GDFR [14] is a collaborative effort between Harvard University, the US National Archives, and the Online Computer Library Center (OCLC). GDFR is also designed for digital preservation to provide sustainable services that store, discover, and deliver information about digital formats.

2.3 Other Related Work

2.3.1 File Mapping

Conti *et al.* [15] recently published an technique that takes a large binary data object such as a memory dump of a running system and uses primitive file types to perform classification of all the potential file types within the data object. The work demonstrates a radically different approach to file fragment identification because it seeks to classify primitive data types within files or a data object (equating to regions within the file/object) rather than trying to classify an entire file or fragment.

For their experiment the authors created statistical signatures of filetypes using 14,000 file fragments from primitive files such as compressed, encrypted, machine code, text, and bitmap files. The authors then compared these signatures to a test set of file fragments using k-nearest neighbor with Euclidean distance as the distance measurement. Their initial accuracy results for certain types were not too impressive. For example, they found their accuracy with encrypted data (AES256/text) was only 38.6% or compressed data such as JPEG was only 44.1% accurate. However, if they grouped the primitive filetypes by similarity (e.g., all random, encrypted, and compressed data), then the accuracy dramatically improved to 98.55%. It was from these observations that lead them to classify regions within a file/object into categories with the results of each distinct region being optionally passed back to the user.

Also of note is the author's *byteplot* utility that graphically displays an object's structure. Each byte value within an object is sequentially plotted on a vertically oriented display. Each pixel value reflects the byte value with 0 being black and 255 being white. This tool graphically allows manual inspection of a plot of a data object where an analyst can click on the regions within the data object to more closely examine, allowing for manual filetype identification.

2.3.2 Summarized Patterns

Summarized pattern features[16] were introduced by Collins [17] in 2002 to assist in named entity recognition. Pattern features are used to map words into a set of patterns over a character class. For example, all uppercase characters could be mapped to the character “A,” all lowercase characters to “a,” and all digits mapped to “5.” Further, these patterns can be condensed [16] into smaller summarized patterns by mapping a continuous string of common characters to a single character (i.e.,: “Room-238” mapped to “Aa-5”).

Summarized patterns can be applied to filetype identification due to the nature of repeating structures within file formats such as designating file versions in a header/footer or indicating embedded objects within the data stream. Within filetype identification, n-grams that are dissimilar may still be indicative of an “entity” that is a feature of the filetype. By applying summarized patterns, dissimilar n-grams can be condensed into a single n-gram that represents all n-grams within a file that refer to the same feature.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Long N-Gram Innovation

It is the hypothesis of this thesis that long n-grams exist as structures within files and can be used to distinctly identify the file's type. Short n-grams of one or two bytes, although they may be important to the structure of a filetype, are not excellent predictors of filetype because they are more likely to appear in random data. However, with each byte added to an n-gram, the likelihood that the n-gram will appear in random data decreases exponentially. Therefore, if a filetype can be characterized by the presence of distinct long n-gram(s), then these n-grams become features describing the filetype of a file or file fragment.

3.1 N-Grams in the Wild

In the following subsections, various filetypes are presented in a split-hex view in order to discuss how filetypes may or may not contain long n-grams that can be used to characterize them. Each split-hex view is abridged and only shows the necessary components of the file that pertain to the discussion.

3.1.1 HTML Source Code

Figure 3.1 shows the contents of a sample HTML source code file. HTML source code is a textual file containing the code that is interpreted by a HTML browser when visiting a webpage. Within this file are HTML tags. The beginning of the HTML file in Figure 3.1 states that the HTML source code conforms to the W3C (World Wide Web Consortium) HTML 4.01 standard. In order for this HTML file to be compliant, the file must contain at the very least the HTML element and the document element. Within the document element, the TITLE element must be present. The beginning and ending of these elements are marked by tags of the form <*start_element*> and <*/close_element*>. The content in between these tags make up the content of the element.

Inspection of Figure 3.1 shows that the HTML file contains the HTML, TITLE, BODY, and other HTML 4.01 tags. On line 0060 is the tag marking the beginning of the HTML code (`<html>`). Immediately following the beginning of the HTML code is the tag marking the beginning of the document head, marked by the tag `<head>` on line 0070. The HTML title is marked by the tag `<title>` on line 00c0. The beginning visible content that will be displayed

```
0000   0a 3c 21 44 4f 43 54 59 50  45 20 48 54 4d 4c 20   .<!DOCTYPE HTML
0010   50 55 42 4c 49 43 20 22 2d  2f 2f 57 33 43 2f 2f   PUBLIC "-//W3C//
0020   44 54 44 20 48 54 4d 4c 20  34 2e 30 31 20 54 72   DTD HTML 4.01 Tr
0030   61 6e 73 69 74 69 6f 6e 61  6c 2f 2f 45 4e 22 0a   ansitional//EN".
0040   22 68 74 74 70 3a 2f 2f 77  77 77 2e 77 33 2e 6f   "http://www.w3.o
0050   72 67 2f 54 52 2f 68 74 6d  6c 34 2f 6c 6f 6f 73   rg/TR/html4/loos
0060   65 2e 64 74 64 22 3e 0a 0a  3c 68 74 6d 6c 20 6c   e.dtd">..<html l
0070   61 6e 67 3d 22 65 6e 22 3e  0a 3c 68 65 61 64 3e   ang="en">.<head>
.
.
.
00c0   2d 38 22 3e 0a 3c 74 69 74  6c 65 3e 20 57 68 65   -8">.<title> Whe
.
.
.
0110   61 6e 64 20 31 39 39 34 2e  3c 2f 74 69 74 6c 65   and 1994.</title
0120   3e 0a 0a 3c 6c 69 6e 6b 20  72 65 6c 3d 22 73 74   >..<link rel="st
.
.
.
01c0   3e 0a 0a 0a 3c 73 63 72 69  70 74 20 74 79 70 65   >...<script type
01d0   3d 22 74 65 78 74 2f 6a 61  76 61 73 63 72 69 70   ="text/javascrip
01e0   74 22 20 73 72 63 3d 22 2f  70 61 67 65 54 65 6d   t" src="/pageTem
01f0   70 6c 61 74 65 73 2f 67 65  6e 65 72 61 6c 2f 67   plates/general/g
0200   77 2e 6a 73 22 3e 0a 3c 2f  73 63 72 69 70 74 3e   w.js">.</script>
.
.
.
0340   74 69 6f 6e 3a 20 6e 6f 6e  65 3b 0a 7d 0a 3c 2f   tion: none;.}.</
0350   73 74 79 6c 65 3e 0a 3c 2f  68 65 61 64 3e 0a 0a   style>.</head>..
0360   3c 62 6f 64 79 20 6f 6e 6c  6f 61 64 3d 22 64 6f   <body onload="do
0370   63 75 6d 65 6e 74 2e 47 4d  42 61 73 69 63 53 65   cument.GMBasicSe
0380   61 72 63 68 2e 55 73 65 72  53 65 61 72 63 68 54   arch.UserSearchT
0390   65 78 74 2e 66 6f 63 75 73  28 29 22 3e 0a 0a 0a   ext.focus()">...
.
.
.
2cb0   0a 0a 3c 2f 66 6f 72 6d 3e  0a 3c 2f 62 6f 64 79   ..</form>.</body
2cc0   3e 3c 2f 68 74 6d 6c 3e 0a  0a                     ></html>..
```

Figure 3.1: Contents of a sample HTML source code

on the webpage is marked with the `<body>` tag on line 0360. All tags associated with the end of the HTML code (`</html>`), the end of the TITLE (`</title>`), and the end of the document (`</body>`) are also present on lines 2cc0, 0110, and 2cb0 respectively.

The presence of all of these tags are indicative of an HTML source code file. These tags are

```
0000   ff d8 ff e0 00 10 4a 46 49  46 00 01 01 01 00 48   ......JFIF.....H
0010   00 48 00 00 ff e2 0c 58 49  43 43 5f 50 52 4f 46   .H.....XICC_PROF
.
.
.
1340   1d 2f c0 47 e4 af ad 86 a6  ff 00 da 8a cd 1d 93   ./.G............
1350   82 fd de 6d 9f 47 b1 71 fa  09 94 4f 62 62 63 f4   ...m.G.q...Obbc.
1360   13 47 a5 fe 47 c3 db b5 81  a5 ee 45 7f 21 6c f9   .G..G......E.!l.
.
.
.
1410   c6 29 ff 00 12 35 34 b6 47  1e 2e 9a 7a d5 bc f0   .)...54.G...z...
.
.
.
1460   ab ff 00 02 32 ab a7 87 3b  98 ee 53 8a be 68 df   ....2...;..S..h.
.
.
.
1540   df 15 6e 46 1c aa a9 56 59  94 a9 f3 71 45 91 9e   ..nF...VY...qE..
1550   66 ef 2b cd bb 27 ff 00 c8  da aa ea b2 37 74 7d   f.+..’.......7t}
1560   5a b0 5e e1 2a 50 fb 45 f7  98 75 f3 53 9d a7 04   Z.^.*P.E..u.S...
.
.
.
15c0   1a df 94 27 7f 45 5f 97 71  28 ed 19 47 84 51 3b   ...’.E_.q(..G.Q;
15d0   2c ac b5 6c c6 59 ff 00 f9  18 50 da 8d 7a 54 62   ,..l.Y....P..zTb
.
.
.
2360   50 d3 ff 00 40 06 64 bf 5e  2d c5 a7 d3 6c 7a 43   P...@.d.^-...lzC
2370   ce af 4d 72 e5 f9 6b 4e 5e  4b 47 e0 3a ce d5 da   ..Mr..kN^KG.:...
2380   b8 dd b7 b4 aa ed 1d a3 5b  7d 8a ad 97 3d 4c 91   ........[}...=L.
2390   8d ec 94 56 91 49 70 48 03  26 f6 9e 53 2d ae 9d   ...V.IpH.&..S-..
23a0   2b 3c 6b 1c 1f ff d9                               +<k....
```

Figure 3.2: Contents of a sample JPEG image

long n-grams and can be used as classification features of an HTML file.

3.1.2 JPEG

Figure 3.2 shows the contents of a sample JPEG file. A valid JPEG file will conform to the JFIF file specification and contains 2-byte markers within the file to annotate certain parameters of the image data [18]. For example, the first two bytes on line 0000 contain the marker 0xFF 0xD8 which is the marker marking the start of the image (SOI). Following those two bytes is another 2-byte marker `0xFF` `0xE0` marking the beginning of the first JPEG Application Segment (APP0) in the file which contains the JFIF metadata. While different JPEG images may very well have different metadata, there will always be a distinct long n-gram within the APP0 segment: the null terminated string "JFIF" shown on line four that is a 5-gram [19]. The SOI marker and the APP0 segment are what the Unix file command utilize to identify a JPEG file. A JPEG file also has an end of image (EOI) marker (`0xFF` `0xD9`) located in the last byte of the image (line 23a0).

The data that makes up the image in a JPEG file is Huffman coded, making the data high entropy data. Within this high entropy data, it is likely that the byte value `0xFF` will occur. However, according to the JPEG specification, the byte value `0xFF` will be interpreted as the firt byte of a JPEG marker by any JPEG capable decoder. In order to prevent framing errors of the image, every byte value of `0xFF` occurring in a Huffman-coded segment is always followed by the byte value 0x00 [18]. When a JPEG encoder encounters the `0xFF` byte value while inside a Huffman-coded segment, the following `0x00` byte value is ignored. This process is termed *byte stuffing* in the JPEG specification. Examples of this can be seen in Figure 3.2 on lines 1340, 1410, 1460, 1550.

Although the segment markers are bigrams and could possibly appear in random data, they appear significantly more often in JPEGs than chance would predict [9], allowing a JPEG fragment to be readily distinguished from random or otherwise compressed data. The JPEG header is sufficiently long enough to help discriminate JPEG data from random data. In this case, the presence of short n-grams such as `0xFF` `0x00` that are descriptive of a filetype are augmented by the presence of a distinctive long n-gram for the filetype. In this manner, whole JPEG files can easily be identified. However, if a JPEG fragment is encountered that does not include the file header, then long n-gram analysis will only have the `0xFF` `0x00` byte stuffing bigram structures to differentiate it from other filetypes.

3.1.3 BMP Image

Figure 3.3 shows the contents of a BMP file. The header information for a BMP file is 14 bytes long. The first two bytes denote the magic number used to identify the type of the BMP file. In figure four, the ASCII characters "BM" can be seen in the first two bytes of the file. These two bytes signify that the BMP file is a Windows BMP file. The next four bytes on the BMP header equal the size of the file (little-endian format) in bytes. In the case of the file showed in Figure 3.3, the file size is 0xAEEC0000-bytes (little-endian) or 60,590-bytes. The next four

```
0000   42 4d ae ec 00 00 00 00 00  00 36 00 00 00 28 00   BM........6...(.
0010   00 00 7d 00 00 00 a1 00 00  00 01 00 18 00 00 00   ..}.............
0020   00 00 78 ec 00 00 13 0b 00  00 13 0b 00 00 00 00   ..x.............
0030   00 00 00 00 00 00 76 55 2b  92 69 35 91 69 38 92   ......vU+.i5.i8.
0040   6a 39 92 6a 35 92 6a 35 92  6a 35 92 69 35 92 69   j9.j5.j5.j5.i5.i
0050   35 91 68 36 91 68 38 91 68  38 90 69 37 90 68 37   5.h6.h8.h8.i7.h7
0060   90 68 37 90 68 39 8f 67 3a  8d 66 39 8c 66 39 8c   .h7.h9.g:.f9.f9.
0070   66 39 8b 65 38 8b 65 3b 8b  65 3b 89 64 39 87 63   f9.e8.e;.e;.d9.c
0080   39 87 62 3b 85 61 39 85 61  39 85 61 3a 84 60 3c   9.b;.a9.a9.a:.`<
0090   82 5f 3b 81 5f 3a 81 5f 3b  80 5f 3b 7e 5d 3b 7d   ._;._:._;._;~];}
.
.
.
ec90   49 2c 64 49 2c 65 4a 2d 65  4a 2d 65 4a 2d 65 4a   I,dI,eJ-eJ-eJ-eJ
eca0   2d 65 4a 2d 66 4a 2d 66 4a  2c 52 3d 22 75         -eJ-fJ-fJ,R="u
```

Figure 3.3: Contents of a sample BMP file

bytes after the BMP file size are reserved for application usage. The file used to generate Figure 3.3 did not have any application data located in this offset and therefore contain zero values. Lastly, the last four bytes of the BMP header give the offset of the beginning of the pixel array within the file itself. This array is the contents of the image itself, with each entry in the array representing individual values of pixels within the image. Since the BMP file shown in figure four is identified as a Windows type of BMP file, each pixel in the image is represented by three bytes: one byte each for a red value, a green value, and a blue value. The pixel array is the last element in a BMP file, no file footer is present for a BMP file.

The BMP file format presents a very difficult problem for long n-gram filetype identification. There are no long n-grams associated with the structure of a BMP file that can be relied upon to accurately describe the characteristics of all BMP files. Although all BMP files have a 14-byte long header, only the first two characters in the header are indicative of *all* BMP files; the remaining bytes are used to encode other information about the file, such as its size and pixel depth. Because different BMP files are sure to have different sizes, characterizing BMP files on specific sizes may not be useful. Therefore, the long n-gram approach will not be effective when applied to BMP files or other formats with similar structure. Indeed, the only way to reliably identify a BMP file is to see if the file size coded in the header matches the length of the file. There is no obvious way to identify a BMP fragment other than rendering the image and seeing if it looks correct.

3.1.4 PDF

Figure 3.4 shows the complete contents of a simple PDF file. A PDF file consist of four parts: a header containing an ASCII string denoting the version of the PDF file (i.e., "%PDF-1.4" on line 0000), a body containing one or more objects in use by the file, a cross reference table of objects in the file, and a trailer. Each object in the body is labeled sequentially in the form: "1 0 obj", "2 0 obj", etc.. The end of objects are marked with an "endobj" (see line 0120-0130). The cross reference table holds all the offsets of all the objects contained within the file for quick lookups and is marked by "xref". The cross reference table can be seen starting on line 17b0 of Figure 3.4. The trailer contains the information regarding where the document starts.

```
0000   25 50 44 46 2d 31 2e 34 0a  25 d0 d4 c5 d8 0a 33   %PDF-1.4.%.....3
0010   20 30 20 6f 62 6a 20 3c 3c  0a 2f 4c 65 6e 67 74    0 obj <<./Lengt
0020   68 20 31 30 32 20 20 20 20  20 20 20 0a 2f 46 69   h 102       ./Fi
0030   6c 74 65 72 20 2f 46 6c 61  74 65 44 65 63 6f 64   lter /FlateDecod
0040   65 0a 3e 3e 0a 73 74 72 65  61 6d 0a 78 da 25 8b   e.>>.stream.x.%.
.
0090   b9 7d cc 76 6c fb 5f ce 25  f1 b9 61 b6 e8 00 14   .}.vl._.%..a....
00a0   18 d9 11 e4 21 0a 26 f9 67  fa 92 e9 d4 bc 2e d4   ....!.&.g.......
00b0   18 91 0a 65 6e 64 73 74 72  65 61 6d 0a 65 6e 64   ...endstream.end
00c0   6f 62 6a 0a 32 20 30 20 6f  62 6a 20 3c 3c 0a 2f   obj.2 0 obj <<./
00d0   54 79 70 65 20 2f 50 61 67  65 0a 2f 43 6f 6e 74   Type /Page./Cont
.
0110   37 36 20 38 34 31 2e 38 39  5d 0a 2f 50 61 72 65   76 841.89]./Pare
0120   6e 74 20 35 20 30 20 52 0a  3e 3e 20 65 6e 64 6f   nt 5 0 R.>> endo
0130   62 6a 0a 31 20 30 20 6f 62  6a 20 3c 3c 0a 2f 46   bj.1 0 obj <<./F
0140   6f 6e 74 20 3c 3c 20 2f 46  38 20 34 20 30 20 52   ont << /F8 4 0 R
.
0360   20 35 30 30 20 35 35 35 2e  36 20 32 37 37 2e 38    500 555.6 277.8
0370   20 33 30 35 2e 36 20 35 32  37 2e 38 20 32 37 37    305.6 527.8 277
0380   2e 38 20 38 33 33 2e 33 20  35 35 35 2e 36 20 35   .8 833.3 555.6 5
0390   30 30 20 35 35 35 2e 36 20  35 32 37 2e 38 20 33   00 555.6 527.8 3
03a0   39 31 2e 37 20 33 39 34 2e  34 5d 0a 65 6e 64 6f   91.7 394.4].endo
.
0400   46 6c 61 74 65 44 65 63 6f  64 65 0a 3e 3e 0a 73   FlateDecode.>>.s
0410   74 72 65 61 6d 0a 78 da ad  92 79 38 94 ed db c7   tream.x...y8....
0420   6d 83 c8 ae 6c e1 46 48 96  31 b2 2f d9 f7 6c 63   m...l.FH.1./..lc
.
14a0   48 33 52 78 8f 39 e1 1d 4a  f3 45 06 83 f9 ff 00   H3Rx.9..J.E.....
14b0   15 25 25 38 0a 65 6e 64 73  74 72 65 61 6d 0a 65   .%%8.endstream.e
14c0   6e 64 6f 62 6a 0a 38 20 30  20 6f 62 6a 20 3c 3c   ndobj.8 0 obj <<
14d0   0a 2f 54 79 70 65 20 2f 46  6f 6e 74 44 65 73 63   ./Type /FontDesc
14e0   72 69 70 74 6f 72 0a 2f 46  6f 6e 74 4e 61 6d 65   riptor./FontName
14f0   20 2f 5a 47 52 54 50 4f 2b  43 4d 52 31 30 0a 2f    /ZGRTPO+CMR10./
1500   46 6c 61 67 73 20 34 0a 2f  46 6f 6e 74 42 42 6f   Flags 4./FontBBo
.
17a0   2e 35 2e 37 29 0a 3e 3e 20  65 6e 64 6f 62 6a 0a   .5.7).>> endobj.
17b0   78 72 65 66 0a 30 20 31 31  0a 30 30 30 30 30 30   xref.0 11.000000
17c0   30 30 30 30 20 36 35 35 33  35 20 66 20 0a 30 30   0000 65535 f .00
17d0   30 30 30 30 30 33 30 37 20  30 30 30 30 30 20 6e   00000307 00000 n
.
1870   30 30 30 30 35 37 36 30 20  30 30 30 30 30 20 6e   00005760 00000 n
1880   20 0a 30 30 30 30 30 30 35  38 30 39 20 30 30 30    .0000005809 000
1890   30 30 20 6e 20 0a 74 72 61  69 6c 65 72 0a 3c 3c   00 n .trailer.<<
18a0   20 2f 53 69 7a 65 20 31 31  0a 2f 52 6f 6f 74 20    /Size 11./Root
18b0   39 20 30 20 52 0a 2f 49 6e  66 6f 20 31 30 20 30   9 0 R./Info 10 0
.
1910   3e 0a 73 74 61 72 74 78 72  65 66 0a 36 30 36 34   >.startxref.6064
1920   0a 25 25 45 4f 46 0a                               .%%EOF.
```

Figure 3.4: Contents of a sample PDF file

The PDF body can contain many types of data like: numbers, arrays, string objects, dictionary, and streams. A dictionary is an unordered list of key-value pairs (name-object) that are hash tables. Example keys are "`/Type`" (line 00c0 and 00d0) and "`/Font`". The stream object is interesting because within a stream in a PDF, embedded data of any type can be stored. PDF files typically embed JPEGs for example. However, they can also embed other files as well, even other PDFs! If an object is a stream, it begins with a dictionary that describes the item such as length or encoding. The actual data stream is marked with "`stream`" and ends with "`endstream`" like on lines 0040 and 00b0 of figure five.

The cross reference table (beginning on line 0x17b0) contains one or more offsets of the objects within the PDF file. Each object listing has three sections: a 10 digit ASCII offset from the beginning of the file to the object's location, a five digit ASCII generation number, and a single ASCII character marking the object as free or in use ("`f`" or "`n`" respectively). In between each string is an ASCII space. The very first object entry in the reference table (lines 17b0-17c0) is the location of the first object marked as free in the document. This entry stores the free object's offset and will always contain the ASCII generation number of "`65535`" and will always have the ASCII character "`f`" at the end. The remaining entries in the reference table will have values associated with the location and status of the object and will vary from document ot document. The final free object listed in the cross reference table will always have the 10 digit ASCII string "`0000000000`" since there are no further free objects within the PDF file.

Lastly, there is the trailer section of a PDF. This section is marked by "`trailer`" (line 1890) and among other various information contained within this part of the PDF file is the offset from the beginning of the file where the cross-reference table begins. This offset is located immediately before the end of file marker for the file which is "`%%EOF`".

The PDF file format is a kind of container file. A container file is a file that has the capability of embedding data of a different type within its structure. For example, many PDF files that contain graphics contain embedded JPEG files. Another example is compressed objects within a PDF file. In order to keep the size of a PDF file to a minimum, encapsulated objects within a PDF file can be compressed. Figure 3.4 shows an example of an embedded object that has been compressed. Starting on line 0000, an object is specified (object "`3 0 obj`"). Following the new object specification, line 0030 contains the string "`FlateDecode`". This informs the application that may be handling the PDF file that the datastream following this string is compressed and can be uncompressed using the *zlib* algorithm. This compressed datastream continues until the “`endstream.endobj`” is specified (line 00b0).

The PDF file format is an excellent example of a filetype that has distinct long n-grams within the structure of the entire file, making the filetype easily identifiable. Further, since the PDF format has a distinct structure marked by distinct long n-grams, file fragments originating from PDF files have a better chance of being accurately identified, since the structuring data is distributed throughout the file format. However, if the file fragment originates from the middle of an encapsulated object, then accurate identification can be significantly impacted by the data type of the embedded object. For example, if the file fragment originates from a JPEG image encapsulated within a PDF file, there is no way to determine if the file fragment is part of a PDF file or a standalone JPEG image without other fragments from the file.

3.2 Algorithm for Finding Long N-Grams

A method to detect common n-grams amongst a set of files is straightforward: take each long n-gram of each n-size from a file and search for that n-gram in all the other files in a set. If the n-gram appears in all the files in the set, then it can be called a distinct n-gram for that particular filetype and added to a set of identifying n-grams. Once all n-grams of each n-size has been searched among the set of files, a generated list of n-grams of various sizes that were present in all the files in the set can be created. This set of n-grams can then be used to check if an unknown file has any of the distinct n-grams contained within it in order to be used in a classification process. The classification process would have to decide if enough of the n-grams listed for the filetype are contained within the unknown file in order to classify that file as being of that filetype. Since long n-grams can be used as features, machine learning techniques can be employed to classify files based on the presence of long n-grams. However, if long n-grams can be determined to be distinct to a single filetype then there really is no need for any sophisticated machine learning since the presence of distinct long n-gram is by itself a strong indicator of a specific filetype.

Long n-grams will be found within a filetype as long as it is common to the filetype. For example, within properly formed HTML files, one would expect to find $\boxed{\texttt{<}}\boxed{\texttt{H}}\boxed{\texttt{T}}\boxed{\texttt{M}}\boxed{\texttt{L}}\boxed{\texttt{>}}$ and $\boxed{\texttt{<}}\boxed{\texttt{/}}\boxed{\texttt{H}}\boxed{\texttt{T}}\boxed{\texttt{M}}\boxed{\texttt{L}}\boxed{\texttt{>}}$ ASCII n-grams since those ASCII strings are part of the HTML language. For JPEG files, the ASCII n-gram $\boxed{\texttt{J}}\boxed{\texttt{F}}\boxed{\texttt{I}}\boxed{\texttt{F}}$ and the n-grams $\boxed{\texttt{0xFF}}\boxed{\texttt{0x00}}$, $\boxed{\texttt{0xFF}}\boxed{\texttt{0xD8}}$, and $\boxed{\texttt{0xFF}}\boxed{\texttt{0xD9}}$ should always be detected within JPEG files.

3.3 Summarized N-Grams

The principle of summarized patterns proposed by Collins to improve named entity extraction [17] can be applied to long n-gram analysis by taking n-grams within a filetype that are representing common features of a filetype and associating them by a single n-gram. The benefit to this is being able to represent repeating or similar structures within a filetype that may vary by a few bytes but are still a single distinctive characteristic. These can be represented by a single summarized n-gram.

3.3.1 Presummarization of N-Grams

Table 3.1 lists common n-grams to nine different filetypes. The common n-grams were found by using learning sets containing 20 files each of a single type. In order for an n-gram to be considered "common", it had to have appeared within every file of a learning set.

Type	Total Found	Example N-gram (HEX)	ASCII Equivalent	N-gram size	Predictive (yes/no)
ppt	1942	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....	20	no
		ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff	....	20	no
		53 00 75 00 6d 00 6d 00 61 00 72 00 79 00 49 00 6e 00 66 00	S.u.m.m.a.r.y.I.n.f.	20	yes
		20 50 6f 77 65 72 50 6f 69 6e 74	.PowerPoint	11	no
		ff 00 ff	...	3	no
doc	108	20 44 6f 63 75 6d 65 6e 74	.Document	9	no
		4d 69 63 72 6f	Micro	5	no
		61 74 69 6f 6e	ation	5	no
pdf	189	2f 43 72 65 61 74 69 6f 6e 44 61 74 65	/CreationDate	13	yes
		30 30 30 30 30 30 30 30 30 30 20	0000000000.	11	yes
		31 30 20 30 20 6f 62 6a	10.0.obj	8	yes
		31 31 20 30 20 6f 62 6a	11.0.obj	8	yes
		25 50 44 46 2d 31 2e	%PDF-1.	7	yes
		20 31 30	.10	3	no
		20 20	..	2	no
gif	3	2c 00 00 00 00	,....	5	no
		47 49 46 38	GIF8	4	yes
		00 3b	.;	2	no
jpg	30	ff d8 ff e0 00 10 4a 46 49 46 00 01	......JFIF..	12	yes
		01 01 00 00 00 00 00 00 00	....	9	no
		ff 00	..	2	no
gz	44	1f 8b 08	...	3	yes
		00 99	..	2	no
		71 84	q.	2	no
		d0 c0	..	2	no
		f9 6f	.o	2	no
ps	125	25 25 45 6e 64 43 6f 6d 6d 65 6e 74 73	%%EndComments	13	yes
		65 66 6f 6e 74 20 73 65 74 66 6f 6e 74	efont.setfont	13	yes
		25 21 50 53 2d 41 64 6f 62 65 2d	%!PS-Adobe-	11	yes
		20 31 30	.10	3	no
		20 20	..	2	no
html	156	20 74 68 65 20	.the.	5	no
		61 62 6c 65 20	able.	5	no
		74 69 6f 6e 20	tion.	5	no
		3e 3c 2f	></	3	no
		20 20	..	2	no

Table 3.1: Sample common n-grams from learning sets of 20 files each. Non-printable ASCII values in ASCII column are displayed as "."

Table 3.1 includes a column named “Predictive (yes/no)”. This column was manually created for this thesis based on intuition as to whether each n-gram would be a useful predictor of a filetype. The classification of an n-gram’s predictability is based on whether I felt that the n-gram was likely to occur within random data or be likely to appear within other filetypes. It should be noted that while the “predictive” column was generated manually, the rest of the table was generated automatically. The final solution to the file identification problem presented later does not require manual intervention.

The PPT filetype in Table 3.1 produced some very long common n-grams. However, the two 20-grams that are all 0x00 values and all `0xFF` values are predictors of the PPT filetype because of their low entropy. Runs of single values are commonly seen in many filetypes. For example, runs of byte values consisting of all 0’s or all 1’s frequently occur in bitmap image files (an all black or an all white BMP file). Additionally, hard drives could also be initialized with all 0’s or all 1’s on them that would produce fragments containing these values. The 20-gram `S.u.m.m.a.r.y.I.n.f` listed for the PPT filetype is classified as a distinct predictor because the 20-gram is a Unicode string that is not a commonly occurring word. In fact, the majority of the 20-grams found in the PPT learning set were a mixture of commonly occurring Unicode strings (low predictability) and uncommon Unicode strings (high predictability) or 20-grams filled with substrings of 0’s followed by 1’s or 1’s followed by 0’s. The the 11-gram and the 3-gram in Table 3.1 are not good predictors for the PPT filetype since the 11-gram is a string of ASCII characters that can occur in any filetype that contains ASCII text. The 3-gram is closely related to the JPEG byte stuffing marker `0xFF` `0x00` (although JPEGS will typically not contain a 3-gram `0xFF` `0x00` `0xFF`). It is important to note that the PPT learning set generated 1942 common n-grams. Although the PPT learning set produced a large amount of low predictability n-grams, they are still important in the overall description of a file because the presence of a large number of the low predictability n-grams within a file/fragment could still be indicative of the filetype.

The DOC filetype in Table 3.1 shows the best n-grams found within a DOC filetype learning set. As shown, all three examples were low predictors of the filetype since all three are all ASCII strings. The remaining 105 n-grams detected in the DOC learning set were all trigrams and bigrams, making them not useful predictors.

The PDF filetype produced a number of long n-grams that are distinct to PDF files. Although the structures within a PDF file are marked by ASCII strings, they are very distinctive ASCII strings. For instance, the 13-gram shown for the PDF filetype in Table 3.1 begins with a forward slash followed by the ASCII string “`CreationDate`”. Although this string may not be very distinct since it contains two dictionary words, concatenating the words together utilizing camel case and the addition of the forward slash at the beginning make this n-gram indicative of PDF files. It is important to note that this n-gram could also originate from a textual file describing the structuring of a PDF file. However, as was discussed in the previous subsection on PDF files and the example n-grams shown in Table 3.1, there are many distinct long n-grams that are part of the PDF file specification that can help to make this ambiguity more clear.

The GIF learning sets produced exactly three n-grams, two of which were of low predictability. The 4-gram originated from the headers of every file in the learning set. Although it is classified as being of high predictability, a GIF file fragment from the middle of the file will not contain this n-gram since it occurs in the header of the file.

The JPEG learning sets produced one distinct common 12-gram. However, like the GIF filetype, it originates from the JPEG file header. The remaining n-grams for the JPEG filetype are low predictors and are mainly trigrams and bigrams. The trigrams and bigrams contained additional JPEG markers such as: Start of Scan (SOS), Define Huffman Table (DHT), and Define Quantization Table (DQT). Although JPEG markers that can be located within high entropy are short n-grams, they can still be useful in identifying JPEG file fragments when the presence of multiple bigrams corresponding to JPEG markers are detected within high entropy data.

The gzip (GZ) filetype produced one distinct n-gram, the trigram `0x1F` `0x8B` `0x08`, from the header of each GZ file in the learning set. This is the header as specified in the GZIP standard (RFC1952). The tri-gram is part of the GZ header specified in RFC1952 for the GZ file specification: `0x1f` `0x8b` (called ID1 and ID2 respectively) identify the file as a gzip file and the `0x08` is the CM field (compression method). This field specifies the compression method used to compress the data. CM1-7 are reserved and CM8 denotes the *deflate* compression method. Although this n-gram is relatively small, its byte values help to make it distinct. The trigram `0xFF` `0x00` `0xFF` from the PPT filetype is not a distinct trigram because of its individual values of either all 1’s or all 0’s and are likely to occur in binary data or other filetypes. The trigram for the GZ filetype, although as equally likely to occur in random data as the trigram discussed for the PPT filetype, is still distinct enough to predict a GZ filetype header.

The post-script filetype (PS) shares the same characteristics of the PDF filetype n-grams. Both share a diverse range of long n-grams that are distinct to their particular filetype. Interestingly, the trigrams and bigrams shown for both the PDF filetype and the PS filetype are common to both filetypes, further demonstrating the low predictability rating of both n-grams.

The HTML n-grams warrant special analysis. One would always expect to see the HTML tags <HTML> and </HTML> in every HTML file. However, the results for the HTML filetype in Table 3.1 do not show the respective n-grams and in fact, they were not present once all 20 HTML learning files were scanned. The reason for this is that the HTML language is case insensitive, meaning <html> and <HTML> have the same semantics. Upon inspection of the 20 HTML files in the learning sets, it was discovered some files contained all uppercase HTML tags while others contained all lowercase tags, eliminating the possibility of finding common n-grams indicative if filetype. Indeed, the most common n-grams are in fact indicative of the language of the file's content, English

3.3.2 Postsummarization of N-Grams

Table 3.2 was generated using the same learning sets used in Table 3.1 and has the same last column containing a manual classification of the predictability of the n-gram. The summarization method used was to convert all ASCII uppercase to lowercase and all ASCII integers to the ASCII number 5.

The PPT filetype experienced a significant growth in the number of common n-grams when summarized n-grams were utilized. However, the use of summarized n-grams did not appear to produce additional distinct n-grams that could be used to describe the PPT filetype. Of the 2590 n-grams found in the PPT learning set, 1590 of them are bigrams which do not appear to be distinctive enough to characterize PPT bigrams from random data bigrams. Although there are no distinguishing n-grams for the PPT filetype, the large number of non-distinguishing n-grams combined can be utilized to identify the PPT filetype. The full file identification experiment discussed in Chapter 4 highlights this fact.

The DOC filetype, like the PPT filetype, did not experience any improvement in the detection of distinguishable n-grams. Before summarization, none of the DOC files in the learning set had a four digit number in common since no 4-gram containing all ASCII integers was in the common n-gram set. However, the 4-gram $\boxed{5}\boxed{5}\boxed{5}\boxed{5}$ is added to the set of common n-grams for the DOC filetype once n-gram summarization is used. The occurrence of this 4-gram establishes that all 20 DOC files in the learning set had at least a four digit ASCII number within their contents. While this summarized 4-gram is not a good predictor of the DOC filetype, it does illustrate

Type	Total Found	Example N-gram (HEX)	ASCII Equivalent	Size	Predictive (yes/no)
ppt	2590	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....	20	no
		ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff	....	20	no
		73 00 75 00 6d 00 6d 00 61 00 72 00 79 00 69 00 6e 00 66 00	s.u.m.m.a.r.y.i.n.f.	20	yes
		0f 00 04 f0 28 00 00 00 01 00 09 f0 10 00 00 00 00 00 00 00	....(...............	20	no
		35 00 00 00 35 00 00 00 35 00 00 00 35 00 00 00 35 00 00 00	5...5...5...5...5...	20	no
		ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00	....	20	no
		ff ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00	....	20	no
		ff ff ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00	....	20	no
		ff ff ff ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00 00 00	....	20	no
		6f 00 77 00 65 00 72 00 70 00 6f 00 69 00 6e 00 74 00 20 00	o.w.e.r.p.o.i.n.t...	20	no
		6f cf 11 86 ea 00 aa 00 b9 29 e8 00 00 00 00 00 00 00 00 00	o.........).........	20	no
		70 00 00 00 71 00 00 00 72 00 00 00 73 00 00 00 74 00 00 00	p...q...r...s...t...	20	no
		70 00 6f 00 69 00 6e 00 74 00 20 00 64 00 6f 00 63 00 75 00	p.o.i.n.t...d.o.c.u.	20	no
		70 00 6f 00 77 00 65 00 72 00 70 00 6f 00 69 00 6e 00 74 00	p.o.w.e.r.p.o.i.n.t.	20	no
		20 70 6f 77 65 72 70 6f 69 6e 74	.powerpoint	11	no
doc	122	20 64 6f 63 75 6d 65 6e 74	.document	9	no
		6e 6f 72 6d 61 6c	normal	6	no
		6d 69 63 72 6f	micro	5	no
		35 35 35 35	5555	4	no
		77 6f 72 64	word	4	no
pdf	174	35 35 35 35 35 35 35 35 35 35 20 35 35 35 35 35 20 66	5555555555.55555.f	18	yes
		35 35 35 35 35 35 35 35 35 35 20 35 35 35 35 35 20 6e	5555555555.55555.n	18	yes
		2f 63 72 65 61 74 69 6f 6e 64 61 74 65	/creationdate	13	yes
		35 20 35 20 35 35 35 20 35 35 35	5.5.555.555	11	yes
		25 70 64 66 2d 35 2e 35	%pdf-5.5	8	yes
		35 35 20 35 20 6f 62 6a	55.5.obj	8	yes
gif	7	67 69 66 35 35 61	gif55a	6	yes
		2c 00 00 00 00	,....	5	no
		00 3b	.;	2	no
		27 35	’5	2	no
		35 65	5e	2	no
		35 68	5h	2	no
		35 bc	5.	2	no
jpg	702	ff d8 ff e0 00 10 6a 66 69 66 00 01	......jfif..	12	yes
		01 01 00 00 00 00 00 00 00	....	9	no
		01 01 01 00	....	4	no
		35 35 35	555	3	no
gz	161	1f 8b 08	...	3	no
		00 35	.5	2	no
ps	121	25 21 70 73 2d 61 64 6f 62 65 2d 35 2e 35	%!ps-adobe-5.5	14	yes
		25 25 65 6e 64 63 6f 6d 6d 65 6e 74 73	%%endcomments	13	yes
html	176	3c 2f 62 6f 64 79 3e	</body>	7	yes
		3c 2f 68 65 61 64 3e	</head>	7	yes
		3c 2f 68 74 6d 6c 3e	</html>	7	yes
		3c 62 6f 64 79	<body	5	yes
		3c 68 74 6d 6c	<html	5	yes

Table 3.2: Sample common n-grams from learning sets of 20 files each with summarized n-grams. Non-printable ASCII values in ASCII column are displayed as “.”

how n-gram summarization can “extract” a common n-gram from a set of files.

The PDF filetype (along with the HTML filetype discussed below) best illustrates two benefits of n-gram summarization: production of longer distinct n-grams and the summarization of a filetype’s structuring. Table 3.2 lists a 13-gram as an example of a non-summarized PDF n-gram. This 13-gram was the longest n-gram detected within the PDF learning set. However, once n-gram summarization was used, additional distinctive n-grams of significantly longer length were detected. The 18-grams listed for the PDF filetype in Table 3.2 originate from the cross-reference table of the PDF files in the learning set. The reason why these 18-grams were not found in the non-summarized n-gram set is because all the PDF files in the learning set had different cross reference tables with different numerical values of the offsets of the various objects located within each file. By converting all ASCII integers to the character "5", these similar n-grams became common n-grams. The same is true for the 11-gram and the two 8-grams.

The GIF filetype experienced a slight benefit from n-gram summarization. Without n-gram summarization, the longest distinct n-gram was the 4-gram `G I F 8`. With n-gram summarization, this 4-gram expanded into the 6-gram `g i f 5 5 a`, a summarized n-gram for `G I F 8 7 A` and `G I F 8 9 A` To reiterate, the benefit comes from having a longer distinct n-gram that is even less likely to occur in random data than the 4-gram. However, this does not alleviate the difficulty of identifying GIF data fragments that do not contain the GIF header.

The JPEG filetype is interesting from the standpoint of the growth observed of the number of n-grams detected going from non-summarized to summarized n-grams. Before summarization, the total n-grams detected for JPEG files was 30 n-grams. After summarization, this number grew to 702. JPEG data is compressed data which is high entropy data where individual bytes values are independent from one another and the data will appear to be random when viewed in its raw form. Without n-gram summarization, a set of high entropy files will have very little n-grams in common. However, the way that n-gram summarization was applied in this thesis reduced the range of byte values that a single byte could be. Since all ASCII integers were converted to a single character and all uppercase ASCII characters were converted to lowercase, 35 byte values[1] were removed from the "vocabulary" of the individual byte values. This reduction in the byte vocabulary caused a larger degree of commonality among the high entropy data of the JPEG files. This is another case where relatively insignificant n-grams can be combined together to help characterize a filetype. The summarized bigrams for the JPEG filetype had 691 bigrams in common, having a wide range in values of `0x00 0x02` to `0xFF 0xC4`.

The GZ filetype experienced another significant growth of common n-grams like the JPEG filetype since the GZ filetype is also high entropy data.

The PS filetype shared the same improvement characteristics as the PDF filetypes once n-gram summarization was applied.

[1] ASCII integer values converted to a single value removes nine byte values and all English ASCII characters limited to lowercase removes 26 byte values.

The HTML filetype shows another example of summarized n-grams improving the detection of distinctive n-grams within a filetype. In Table 3.1, common HTML tags such as `<HTML>` were obviously missing when they should have been present given the fact that all files in the learning set were valid HTML files. However, as was discussed, the HTML language is case insensitive. N-gram summarization solved this issue and, as Table 3.2 shows, HTML tags (and the corresponding n-grams they produced) were found. It is important to note that the HTML n-grams in Table 3.2 that correspond to HTML tags were the only HTML tags found in the learning set. The HTML language has numerous other tags that are contained in valid HTML code. However, these tags are not required for a basic and valid HTML document and therefore will not be common to all HTML documents. The majority of the n-grams found within the HTML learning set were trigrams and bigrams of printable ASCII characters (156 in total) corresponding to common bigrams and trigrams in the English language. This is unfortunate as the use of such n-grams for type identification is incorrect, since these n-grams correspond to file content, not to filetype.

3.3.3 Long N-Gram Learning Sets

As with any statistical approach that uses sets of common objects to build a general model, it is important to have learning sets of sufficient size and diversity that an algorithm can then utilize to build a model that accurately reflects the distinct properties of whatever it is modeling. However, establishing the proper size of this set for a specific application is research topic in itself. If a set is too small, an algorithm may not be able to produce a model that can accurately represent the general features associated to that object class. If the set is too big, it may generate a model that is too specific and the model may only represent a subset of the object class and not be useful as a general object classifier. This can happen by eliminating features that are indicative to an object but do not necessarily appear in all objects. Therefore, when designing learning sets, consideration needs to be given to how much generality needs to be given to the distinct features that make up the model.

The general assumption about long n-grams in this thesis is the longer a distinct common n-gram is, the better it will be at predicting a filetype (or a class of filetypes if it is distinct to more than one.) Common short n-grams are important since they could be used to differentiate between filetypes that may have long n-grams in common. However, relying solely on shorter n-grams is not reasonable for feature identification because the space of short n-grams makes them likely to occur across filetypes and in high entropy data.

Table 3.3 shows the top 15 n-grams from 3 feature generations (n-size 2 to 20) from a set of

Set Size	Top 15 N-grams (HEX)	ASCII Equivalent	N-gram Size
20	20 30 30 30 30 30 20 6e 0d 0a 74 72 61 69 6c 65 72 0d	.00000.n..trailer.	18
	65 6e 64 73 74 72 65 61 6d 0d 65 6e 64 6f 62 6a 0d	endstream.endobj.	19
	2f 46 6f 6e 74 44 65 73 63 72 69 70 74 6f 72 20	/FontDescriptor.	16
	2f 57 69 6e 41 6e 73 69 45 6e 63 6f 64 69 6e 67	/WinAnsiEncoding	16
	2f 49 74 61 6c 69 63 41 6e 67 6c 65 20 30	/ItalicAngle.0	14
	33 20 30 30 30 30 30 20 6e 0d 0a 30 30 30	3.00000.n..000	14
	0d 65 6e 64 6f 62 6a 0d 78 72 65 66 0d	.endobj.xref.	13
	2f 43 72 65 61 74 69 6f 6e 44 61 74 65	/CreationDate	13
	2f 46 6c 61 74 65 44 65 63 6f 64 65	/FlateDecode	12
	2f 43 61 70 48 65 69 67 68 74 20	/CapHeight.	11
	2f 46 69 72 73 74 43 68 61 72 20	/FirstChar.	11
	30 30 30 30 30 30 30 30 30 30 20	0000000000.	11
	31 20 30 30 30 30 30 20 6e 0d 0a	1.00000.n..	11
	2f 4c 61 73 74 43 68 61 72 20	/LastChar.	10
	2f 4d 65 74 61 64 61 74 61 20	/Metadata.	10
	2f 52 65 73 6f 75 72 63 65 73	/Resources	10
50	2f 46 6c 61 74 65 44 65 63 6f 64 65	/FlateDecode	12
	30 30 30 30 30 30 30 30 30 30 20	0000000000.	11
	2f 52 65 73 6f 75 72 63 65 73	/Resources	10
	31 36 20 30 30 30 30 30 20 6e	16.00000.n	10
	2f 43 6f 6e 74 65 6e 74 73	/Contents	9
	2f 4d 65 64 69 61 42 6f 78	/MediaBox	9
	30 20 30 30 30 30 30 20 6e	0.00000.n	9
	30 30 30 30 30 30 30 30 31	000000001	9
	32 20 30 30 30 30 30 20 6e	2.00000.n	9
	33 20 30 30 30 30 30 20 6e	3.00000.n	9
	35 20 30 30 30 30 30 20 6e	5.00000.n	9
	37 20 30 30 30 30 30 20 6e	7.00000.n	9
	65 6e 64 73 74 72 65 61 6d	endstream	9
	73 74 61 72 74 78 72 65 66	startxref	9
	20 36 35 35 33 35 20 66	.65535.f	8
85	30 30 30 30 30 30 30 30 30 30 20	0000000000.	11
	2f 52 65 73 6f 75 72 63 65 73	/Resources	10
	2f 43 6f 6e 74 65 6e 74 73	/Contents	9
	2f 4d 65 64 69 61 42 6f 78	/MediaBox	9
	30 20 30 30 30 30 30 20 6e	0.00000.n	9
	30 30 30 30 30 30 30 30 31	000000001	9
	33 20 30 30 30 30 30 20 6e	3.00000.n	9
	35 20 30 30 30 30 30 20 6e	5.00000.n	9
	36 20 30 30 30 30 30 20 6e	6.00000.n	9
	65 6e 64 73 74 72 65 61 6d	endstream	9
	73 74 61 72 74 78 72 65 66	startxref	9
	20 36 35 35 33 35 20 66	.65535.f	8
	25 50 44 46 2d 31 2e 34	%PDF-1.4	8
	2f 43 61 74 61 6c 6f 67	/Catalog	8
	2f 4c 65 6e 67 74 68 20	/Length.	8

Table 3.3: Top 15 n-grams from featrue distributions generated using different sized learning sets. Non-printable ASCII values in ASCII column are displayed as "."

85 PDF files taken from the a zipfile from the *govdocs1* corpus [7]. Each zipfile contains 1000 randomly chosen files from the *govdocs1* corpus. The first row contains the top 15 common

n-grams from 20 PDF files that are a subset of the 85 PDF file set. The second row contains the top 15 n-grams from a subset of 50 PDF files and the third row contains the top 15 n-grams from all 85 PDF files in the set.

The paragraphs above describe characteristics that are distinct to the PDF filetype and should be expected to be found within a feature distribution of the PDF filetype. However, some of the characteristics listed will vary from document to document. For example, in Table 3.2 the set with 20 files had the common 18-gram `.00000.n..trailer.`. This is likely to be part of the "trailer" ASCII string marking the trailer sections of the 20 files in the set. However, the 50 and 85 file sets did not have this 20-gram, instead they both had the 7-gram `trailer`[2]. In this case, the larger file sets produced a more distinct n-gram. If the feature distribution generated from the set of 20 learning files was used to try and identify PDF files, the 18-gram `.00000.n..trailer.` may very well not be a great feature since the subgram `.00000.n..` may not always be located directly in front of the `trailer.` subgram. However, the learning set of 20 files did produce distinct n-grams that are very useful long n-grams that can be applied to identification. The n-grams `/FontDescriptor.` and `/WinsAnsiEncoding` appeared in all the files in the set of 20 PDF files. However, these n-grams are identifiers and won't necessarily appear in all PDF files (and indeed, they were not in the 50 file set or the 85 file set.) In this case, the smaller set generated characteristics that are more useful to building a model of characteristics of PDF files.

The size of the sets also impacted the number of n-grams found. The set of 20 files produced 571 n-grams while the set of 50 files and the set of 85 files produced 242 and 101 n-grams respectively. The difference between the numbers of n-grams between the set of 20 files and the 50 and 85 file sets are 57% and 82.3% respectively. The difference between the 50 file set and the 85 file set was 58%. Although the set of 20 files produced the most common and longest features of the files sampled, the longer features (ngrams) may not be as useful to represent all PDF files as the smaller ones they are related to in the other two sets which are more distinct of PDF files. Further, having more files in a set does not necessarily guarantee that features that are characteristic of a filetype will be found. For example, the 12-gram `/FlateDecode` was found in the set of 50 files but not in the set of 85 files. The 12-gram is part of an embedded object that has been compressed. Although not necessarily in every PDF in existence, it is

[2] These 7-grams were not included in Table 3.3 since they were not in the top 15 n-grams but they were present in the respective feature distributions

fairly common to compress some data in a PDF file. In this case, the set of 85 files eliminated characteristics that would have been useful for identification.

The learning sets used in this thesis were all based on sets of 20 files. Although this section demonstrated that the set of 50 files made a suitable choice in some cases for a learning set of PDF files, the same may not be for other types of files. PDF files are container files that clearly have many distinct characteristics that can be used to identify them. However, learning sets of 20 files for other filetypes that are either less sophisticated file containers or are files consisting of a single primitive type may work just fine for those filetypes. The next section discusses summarized n-grams and how they can positively impact n-gram generation.

3.4 Software Development Overview

Filetype identification utilizing n-gram analysis has so far only utilized short n-grams of one or two bytes in size. The total space of short n-grams is relatively small (256 possible unigrams, 65,536 possible bigrams) and sampling all short n-grams in a file will produce a dense distribution of the frequencies of all possible n-grams in the space. Some of the n-grams may not occur within the file. These zero frequencies are important because the lack of certain n-grams could point to a feature that is indicative to a particular filetype within this “small” space of n-grams. However, as the size of an n-gram increases for each byte added, the space increases exponentially. This exponential space increase will cause an increasing number of long n-grams to have zero frequencies. This ever increasing number of zero frequencies within the distribution of the n-gram space will cause the distribution of frequencies to become sparse. The importance of zero frequencies then diminishes as it becomes a dominant feature across filetypes. Popular, naïve machine learning techniques expect the data it is fed to be dense and therefore can not be depended on to produce accurate results with sparse data. One of the reasons why previous implementations have only used short n-grams is due to their implementation techniques. Previous researchers used a single array *a[256]* to hold byte frequencies, a two dimensional array *a[256][256]* to hold bigram frequencies, and proposed using a three dimensional array to hold trigram frequencies. Clearly, this implementation does not scale. The algorithm to create long n-gram analysis described in this thesis first builds a list of commonly occurring n-grams within a set of files rather than keeping track of the frequencies of every n-gram within the space of the n-gram size. Then for each successive file in the set, all of a file’s n-grams are checked to see whether or not the n-gram is contained in the running list. If an n-gram is found to be contained in the list but not within a file, that n-gram is removed from the list. After all files have been processed, the list contains all the n-grams that were found in all of the files. The classification process in this thesis generates distributions of filetypes out of properly labeled learning sets. For each file within a test set, the filetype of a test file was determined by whichever frequency distribution had the highest average of matching long n-grams. From feature generation to classification, this filetype identification algorithm attempts to exploit the sparse nature of the occurrences of long n-grams in files and make predictions of the file or file fragment's type based on those features detected. Clearly more sophisticated classifiers could be used with respect to n-gram features such support vector machines or tree-based classifiers.

The programming language selected was C++ mainly due to the familiarity and preference of the researcher and the Standard Template Libraries' (STL) offering efficient implementations of associative storage containers. The development environment chosen was Linux, kernel version 2.6.32-SMP, and the tools developed were run on a laptop with a dual-core 2.0GHz AMD Turion 64-bit processor with 4GB of RAM.

3.4.1 Software Design

The first application, *featuregen*, takes sets of exemplar files and generates a feature distribution for each set. This application utilizes a binary file format to save the generated distributions to a file that can be accessed at a later time. In order to decrease the amount of n-grams in a distribution, n-gram subset removal was implemented in *featuregen* as a final pruning step. Subset removal works by taking each n-gram within a distribution and compares it to each n-gram that is of size *n+1*. If the n-gram is found within any other n-gram, it is removed from the distribution. For example, if the n-gram $\boxed{\texttt{a}}\boxed{\texttt{b}}\boxed{\texttt{c}}$ is in the distribution and the n-gram $\boxed{\texttt{a}}\boxed{\texttt{b}}\boxed{\texttt{c}}\boxed{\texttt{d}}$ is also present, then $\boxed{\texttt{a}}\boxed{\texttt{b}}\boxed{\texttt{c}}$ will be removed since it is a subset of $\boxed{\texttt{a}}\boxed{\texttt{b}}\boxed{\texttt{c}}\boxed{\texttt{d}}$.

The second application *featurecomp* was developed to take saved distribution data files, reconstruct the distributions in memory, map unknown files into memory, and then autonomously classify the unknown files based on an algorithm. Upon completion, *featurecomp* saves a confusion matrix based on the autonomous classification of the files. Additional details of both applications will be discussed later on in this chapter.

Summarized n-grams were implemented in both applications. Feature distribution generation utilizes summarized n-grams by creating a summarized n-gram from each n-gram taken from a file. Each n-gram is first run through the *tolower()* string method in order to convert all capital ASCII values (if any are present in the n-gram) to their lowercase equivalent. Next, the n-gram is run through a method that converts all ASCII integer characters to the character "5". Upon completion of both methods, the n-gram is then a summarized n-gram and is ready to be inserted or searched for in a dictionary. The same methods are used for *featurecomp*. However, rather than using them on each n-gram in an unknown file, the whole file is fed through each method. After the file has been converted, each n-gram can be analyzed against all dictionaries in order to map features to a filetype.

3.4.2 Feature Distribution Generation – "FeatureGen"

As stated earlier, the main purpose of this application was to take a learning set of files consisting of a single type and construct a feature distribution from the minimum n-size up to a user specified n-size. This application takes a minimum of two command line arguments: the directory containing the set of learning files and the maximum n-size. The maximum n-size argument is required because the application will iterate through the generation algorithm until it reaches the maximum n-size. For the purpose of this thesis, a hard-coded minimum n-size of 2 was chosen because unigrams, while useful for frequency distributions, are not useful for feature distributions utilizing long n-grams.

At run time the application maps the first file within the learning set into memory. It then generates lists of n-grams from n-size 2 up to the user specified maximum n-size. An STL map object is utilized to store each list in key-value pairs: the key being the n-gram stored within a string object and the value being an integer value that has uses for various methods implemented within the application.[3] After generation of each list for an n-size, the map object is placed into an STL vector object. The lists stored in the vector object are used as the "running" list of n-grams.

With each following file in the learning set, each list in the vector object is intersected with the file and placed back into the vector object. Each intersection of the running list and a file contains the n-grams common to the files processed up to that point. Upon completion of the intersection of the last file with the running list, the resulting lists in the vector object contain the n-grams that were common to all the files in the learning set. All together, the lists comprise all the n-grams that are common to the filetype and all together can be described as the feature distribution of the filetype.

As long as it was not disabled at run time, subset removal is performed. This process takes

[3] For clarification, the term "list" shall refer to the STL map object that is being utilized to store lists of n-grams and not the STL list object itself. The STL list object is a doubly liked list and for the purpose of feature distribution generation, the STL map object was better suited.

each n-gram from a list containing n-grams of size *n* and uses the STL string *find()* method and compares the n-gram to each n-gram in the *n+1* sized list. If the *find()* method returns any value greater than or equal to zero (the index in the string where the substring was found) it is removed from the n-sized n-gram list. This method gives a benefit to performance by significantly reducing the overall size of the feature distribution. The smaller the size of the lists, the less string comparisons *featurecomp* needs to do when classifying unknown files. After subset removal, the list data files are saved to disk and a split-hex view text file is outputted to the working directory displaying all the common n-grams from the minimum n-size up to the user specified n-size. If any list in the generated feature distribution ended up being empty then no file for that n-size will be saved. The list data file is a binary file that contains all information about a list of a single n-size that is necessary to reconstruct them in memory. For further details on the structure of the data file, see the source code for details.

3.4.3 Feature Comparison – "FeatureComp"

The *featurecomp* program loads the saved n-gram lists and classifies unknown files. For clarification, the term unknown files refers to files or file fragments that are contained within a test set or files that have not been verified to actually be of the type specified by their file extensions. *featurecomp* implements a simple scoring algorithm. If a file or file fragment contains more features of one feature distribution than any other distribution, then the identification of the file is made based on that.

The first thing *featurecomp* performs at run time is to go to a user specified location that contains all the n-gram list data files that the user wishes to utilize for the classification process and makes a record of all the data files found. For each data file in the record, *featurecomp* groups the records together by the filetype. Next, the individual data files in each data set are then sorted in ascending order based on the n-gram size. Once this process is complete, *featurecomp* has the feature distributions for the filetypes previously observed while gathering the data files.

The next step in the classification process looks in the user specified directory that contains the files to classify. For each file in the unknown file set, the file is mapped into memory. *featurecomp* then performs an intersection between the mapped file and every data file within each feature distribution, in the same manner *featuregen* performed intersections. Before each intersection, the number of n-grams in an n-gram data file is saved. After the intersection, a new n-gram list is generated containing the n-grams that were found within both the n-gram list data file and the unknown file. The number of n-grams in this new list is divided by the number of n-grams saved before the intersection. This gives the precentage of n-grams contained in the n-gram list data file that were also found within the unknown file. This percentage is calculated for each n-gram list within the feature distribution and stored. These percentages are then summed and divided by the amount of n-gram lists in a feature distribution, giving the average of the precent of n-grams present in a file that are part of a feature distribution. The average is now the commonality between a file and a feature distribution. This commonality is then saved and the technique is conducted for each feature distribution that is compared to a file. Once all feature distributions have been intersected with the mapped file, *featurecomp* then assigns the filetype of the file by whichever feature distribution had the highest commonality with the file.

The way *featurecomp* performed the averaging had to be adjusted to accommodate the different numbers of data files within each data set. *featuregen* will not save a data file for an n-gram list that does not contain any n-grams. Therefore, feature distributions can have different amounts of data files within them. If the commonality was computed using only the sum of the n-gram presence percentages divided by the sum of the data files within a data set, then the commonality averages would not be equally weighted for each feature distribution. For example, take two feature distributions A and B; A with only two n-gram list data files (2-grams and 3-grams) and B with three data files (2-grams, 3-grams, and 4-grams). If both data sets had 100% commonality for the 2-grams and 3-grams with a test file that is of type B, and data set B only had 50% commonality with the test file, then the test file would inaccurately be assigned type A since it would have a higher commonality average. However, if a zero percent average was averaged into data set A's commonality average (meaning A has no feature n-grams of size 4), feature distribution B would have the higher commonality average and the test file would accurately be classified. In order to implement this into *featurecomp*, it keeps track of the highest n-size observed for all feature distributions and adds a zero percent to the percentages of matching n-grams whenever a feature distribution is missing a list of n-grams for an n-size.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Experiments

4.1 Axelsson File Set

To test the long n-gram analysis approach described in Chapter 3, I used Axelsson's data set to compare the long n-gram approach to another filetype identification technique. Axlesson conducted 10 trials within his experiment, each with a different set of files from the Garfinkel *et al.* corpus. Axelsson provided me with the lists of files used during his experiments and the offsets of all the file fragments that were generated from these files. For this experiment, the set of files used was from the first trial in the Axelsson experiment, identified by Axelsson as set "NEW-EXP-0", containing 267 files of 28 different filetypes. The 512-byte file fragments were generated by parsing his list of file fragment offsets and grabbing the data from the files with a Python program.

During the course of setting up the experiment, it was discovered that many files in the test set used in Axelsson's experiment were mislabeled. For consistency with Axelsson's work, all of the experiments presented in this chapter were performed with both Axelsson's incorrect labeled set and with a correctly labeled set. The following are some examples of the mislabeled data; 10 out of the 10 JAR files, 8 out of 10 TTF files, 10 out of 10 ZIP files and 8 out of 10 XBM files were either HTML files or changelog text files.

In general, the large n-gram approach outperformed Axelsson's NCD approach even with improperly labeled training data. With correctly labeled training data, the performance was even better. We do not know how well Axelsson's approach would work with correctly labeled data.

4.1.1 Setup

Feature distribution generation over learning sets containing relatively large files (>10MB) take longer amounts of time to generate. Therefore, in order to keep the feature distribution generation time for all 28 filetypes used in the experiment to a minimum and to create representative feature distributions of filetypes, learning sets were built with 20 files each. Summarized n-grams were used during generation in order to maximize the production of distinctive n-grams versus the learning set size.

The intention was to utilize the Garfinkel *et al.* *govdocs1* corpus to build the learning sets.

However, for some filetypes, such as XBM (X BitMap files), there were not enough of the files present within the corpus to build a learning set that would not include files that were also within the test set specified by Axelsson. Therefore, the *govdocs1* corpus was utilized as much as possible but was supplemented with some files gathered from publicly available resources on the Internet. Each file within the learning set was hand verified to be properly labeled and then distributions for each filetype were generated. The PUB filetype (a Microsoft Publisher document associated with MS Office suite) did not end up having any common n-grams among the 20 files in the learning set. Any file or file fragment in the test set that was a PUB file or originated from one was given the classification of UNK unless it shared common n-grams with another filetype, which in that case would be classified as that filetype.

4.2 Experiment 1 – Classification of Whole Files

The first experiment conducted was classification of complete files within the Axelsson NEW-EXP-0 test set. Table 4.1 shows the confusion matrix of the classification results. The accuracy for this experiment was 49%, with an average precision of 51% and average recall 48%. Although those results are not remarkable, there are some interesting results within the confusion matrix.

	Classified As																													
File Type	BMP	CSV	DOC	DOCX	EPS	GIF	GZ	HTML	JAR	JAVA	JPEG	JS	PDF	PNG	PPS	PPT	PPTX	PS	SQL	SWF	TEXT	TTF	UNK	XBM	XLS	XLSX	XML	ZIP	Files	Recall (%)
BMP	8	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	80
CSV	0	0	1	0	0	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0
DOC	0	0	3	0	0	0	0	1	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	7	42
DOCX	0	0	0	8	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	88
EPS	0	0	0	0	10	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	90
GIF	1	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	90
GZ	0	1	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	90
HTML	0	0	0	0	0	0	0	3	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	33
JAR	0	0	0	0	0	0	0	0	0	1	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	0	10	0
JAVA	0	0	0	0	0	0	0	0	0	9	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	90
JPEG	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	100
JS	0	0	0	0	0	0	0	0	0	5	0	3	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	10	30
PDF	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	100
PNG	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	100
PPS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	10	0
PPT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0	9	100
PPTX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	10	100
PS	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0
SQL	0	0	0	0	0	0	0	0	0	4	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	10	0
SWF	3	3	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	10	10
TEXT	0	0	0	0	0	0	0	0	0	6	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	8	0
TTF	0	1	0	0	0	0	0	0	0	4	0	3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	10	0
UNK	0	5	0	0	0	0	0	0	0	4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0
XBM	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	10	20
XLS	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	1	0	9	55
XLSX	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5	0	0	10	50
XML	0	0	0	0	0	0	0	0	0	3	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	6	0	10	60
ZIP	0	0	0	0	0	0	0	0	0	7	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0
Total Classifications	12	10	4	8	15	9	9	7	0	75	10	19	10	11	4	22	10	0	1	1	0	0	0	2	6	5	16	1	267	
Precision (%)	66	0	75	100	66	100	100	42	0	12	100	15	100	90	0	40	100	0	0	100	0	0	0	100	83	100	37	0		

Table 4.1: Axelsson NEW-EXP-0 File Set – full file classification confusion matrix

The JAVA filetype was the most classified filetype with 75 total classifications with a precision of 12.% and a recall of 90%. The precision for identifying JAVA files is low. However the confusion matrix shows the the majority of the false positives for the JAVA filetype come from purely textual based files (i.e.,: CSV, HTML, SQL, JS, etc). For example, the recall for HTML files was 33%: 3 out of 9 correctly classified as HTML with the remaining six files being classified as JAVA. For CSV files, the recall was zero but 9 out of the 10 CSV files were classified as JAVA. For XBM files, the recall was a low 20% (8 out of 10). However, as noted in the introduction for this section, 8 out of 10 XBM files were either HTML files or changelog text files. The XBM files that were not correctly classified (which were actually text based files) were classified as JAVA files. In the same manner, the ZIP file classifications were interesting and, reiterating, all 10 ZIP files in the test set were of a textual filetype (HTML or text file). For the ZIP file classifications, 7 out of the 10 ZIP files were classified as JAVA with the remaining 3 being classified as JS files, another textual based filetype. Although the individual precision for each textual based filetype was low, overall, the long n-gram approach can be said to be effective at recognizing filetypes that are purely textual based. This will be further discussed in a later section where the commonly occurring n-grams among the test set are presented and discussed.

The Microsoft Office files that conform to the Open XML format (i.e.,: DOCX, PPTX, XLSX) performed rather well. All three filetypes had a precision of 100%. However, these filetypes are actually valid ZIP archive files. These files are well known to be high entropy and, as discussed previously in this thesis, high entropy data has the characteristic of random data. Multiple files containing high entropy data will more than likely not share distinct long n-grams. However, the confusion matrix shows the contrary. The ZIP archive file format stores the names of the file(s) and folder(s) it contains in plain text within the ZIP file. The Open XML format uses a very specific naming scheme and archive structure to store the data. These long n-grams are very distinctive and the algorithm used for filetype identification was able to use these long n-grams for high precision identification.

The presence of file headers also contributed to the success of some high entropy filetypes. The BMP, GIF, GZ, JPEG, and PNG files all had relatively good precision and recall rates. These files all have distinct long n-grams located within their respective file headers that enabled these filetypes to be identified with long n-grams.

Table 4.2 lists 10 out of the 28 filetypes in the test set had precisions of 90% or greater. Although

File Type	Precision	Recall
DOCX	1.00	0.89
GIF	1.00	0.90
GZ	1.00	0.90
JPEG	1.00	1.00
PDF	1.00	1.00
PNG	0.91	1.00
PPTX	1.00	1.00
SWF	1.00	0.10
XBM	1.00	0.20
XLSX	1.00	0.50

Table 4.2: Filetypes with greater than 90% precision for whole file classification

only 2 out of the 10 XBM files in the test set were actually XBM files, the results for XBM files are still included in Table 4.2. If the 8 XBM files that were incorrectly labeled were correctly labeled, the recall for XBM files would be 100%. With the exception of the GZ and SWF filetypes, the other filetypes within Table 4.2 contained long and distinct n-grams for their feature distributions. These features were either part of the structuring of the filetype (i.e.,: PDF, JPEG, XBM) and/or related to the header/footer information of the filetype (i.e.,: GIF header). These features enabled the classification method to precisely classify these filetypes based on the presence of these features. Interestingly, the GZ filetype only had 161 total n-gram features in its distribution. The highest n-gram was the 3-gram `0x1f` `0x8b` `0x08` and the remaining bigrams ranging in values from `0x00` `0x35` to `0xfa` `0x35`. The combination of the 3-gram header and the characteristic high entropy data contained within a GZ file producing a wide range of bigrams made the GZ filetype distinguishable even though the structure of a GZ file is simplistic. The SWF filetype, although it produced a 100% precision, only had one correct classification made. The remaining 9 SWF files in the test set were incorrectly classified as either CSV, BMP, HTML, or ZIP.

XLSX had the lowest recall of 50% out of the top performers for this experiment. Five of the ten XLSX files were correctly classified. Four were classified as JAVA and one was classified as XLS. The XLSX format is zipped XML file developed by Microsoft. The XLSX format can be opened outside of an XLSX handling application by unzipping the file using any decompression utility that supports ZIP compression and archiving. The contents of the file is stored in XML and can be browsed using any XML browser. Upon investigation of the low recall rate for the XLSX filtype, it was discovered four of the ten XLSX files were actually HTML files. Further, since one of the XLSX files were supposedly misclassified as an XLS file, that file was not found to be a ZIP file and was in fact an XLS file. With these facts revealed, the recall of the XLSX type is actually 100%.

4.3 Experiment 2 – Classification of File Fragments

Table 4.3 displays the confusion matrix for the 512-byte file fragment n-gram analysis for the Axelsson test set. The accuracy for this experiment was 20%, with an average precision of 27% and average recall 21%. These results are even less remarkable than the full file analysis from experiment one. However, there are interesting parallels from the first experiment that can be observed within this experiment.

	Classified As																													
File Type	BMP	CSV	DOC	DOCX	EPS	GIF	GZ	HTML	JAR	JAVA	JPEG	JS	PDF	PNG	PPS	PPT	PPTX	PS	SQL	SWF	TEXT	TTF	UNK	XBM	XLS	XLSX	XML	ZIP	Files	Recall (%)
BMP	23	15	0	0	0	0	0	6	0	18	0	0	0	21	0	0	1	0	0	2	0	0	9	0	16	0	1	10	122	18
CSV	0	35	0	0	0	0	0	0	0	85	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	122	28
DOC	45	15	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	29	91	0
DOCX	2	3	0	0	1	0	0	0	0	14	0	1	1	3	0	0	0	0	4	0	0	0	0	0	0	0	1	105	135	0
EPS	0	0	0	0	124	0	0	0	0	13	0	0	0	0	0	0	0	24	0	0	0	0	0	0	0	0	0	0	161	77
GIF	15	0	0	0	0	32	0	0	0	0	6	0	0	39	0	0	0	0	7	9	0	0	0	0	3	0	0	21	132	24
GZ	0	18	0	0	2	18	11	0	0	0	1	7	0	8	0	0	0	1	53	15	0	0	0	0	0	1	0	9	144	7
HTML	0	0	0	0	0	0	0	6	0	88	0	0	0	0	15	0	0	0	0	0	0	0	0	0	3	0	23	0	135	4
JAR	0	0	0	0	0	0	0	0	30	36	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	66	45
JAVA	0	0	23	0	0	0	0	0	0	119	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	142	83
JPEG	15	8	0	0	0	9	0	0	1	0	64	0	1	12	0	0	0	0	13	9	4	0	0	0	0	0	0	14	150	42
JS	0	0	0	0	0	0	0	0	0	47	0	0	0	0	10	0	0	0	25	0	0	0	0	0	0	0	0	0	82	0
PDF	0	2	0	0	0	14	0	0	0	0	15	0	80	0	0	0	0	0	6	0	0	0	0	0	33	0	0	0	150	53
PNG	0	0	0	0	0	42	0	10	0	0	0	0	0	40	0	0	2	0	41	1	0	0	0	0	0	0	0	14	150	26
PPS	1	3	0	0	0	12	2	0	0	0	35	0	2	55	0	0	3	0	10	11	1	0	0	0	1	0	0	14	150	0
PPT	23	5	0	0	0	8	1	0	0	0	0	0	0	34	0	1	0	0	7	0	1	0	0	2	5	6	0	42	135	0
PPTX	1	4	0	0	0	6	2	0	0	0	0	0	0	12	3	0	2	0	3	4	0	0	0	0	0	0	0	113	150	1
PS	0	0	0	0	27	0	0	0	0	0	0	0	0	0	0	0	0	42	0	0	0	0	0	0	0	0	6	0	75	56
SQL	0	0	0	0	0	0	0	0	0	71	0	21	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	96	0
SWF	15	32	0	0	0	11	0	2	0	0	15	14	0	14	0	0	0	0	0	5	0	0	0	0	0	0	0	0	108	4
TEXT	0	11	1	0	0	0	0	0	0	54	0	0	0	0	0	0	0	0	15	0	0	0	0	0	15	0	15	0	111	0
TTF	0	15	0	0	0	0	0	0	0	84	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	106	0
UNK	0	5	0	0	0	0	0	0	0	60	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	67	0
XBM	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	0	0	0	57	12
XLS	26	0	0	1	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	59	0	15	14	125	47
XLSX	1	0	0	0	0	0	0	0	0	52	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	2	78	144	0
XML	0	0	0	0	0	0	0	0	0	14	0	3	0	0	4	0	0	0	0	0	0	0	0	0	0	0	89	0	110	80
ZIP	0	0	0	0	0	0	0	0	0	79	0	0	0	0	12	0	0	0	15	0	0	0	0	0	0	0	5	0	111	0
Total Classifications	167	171	24	1	154	152	16	25	31	884	136	46	84	240	59	1	8	67	199	56	6	0	9	9	146	7	166	463	3327	
Precision (%)	13	20	0	0	80	21	68	24	96	13	47	0	95	16	0	100	25	62	0	8	0	0	0	77	40	0	53	0		

Table 4.3: Axelsson NEW-EXP-0 File Set – 512-byte file fragment classification confusion matrix

The JAVA filetype was again the filetype with the most classifications made (884 classifications). Like in the first experiment, most of the false positives for the JAVA type came from file fragments that were from textual based filetypes: CSV, HTML, JS, SQL, TEXT, and XML. The n-gram distribution of the Java files contained 87 n-grams from lengths 2 up to 7 bytes in length. The bi-grams were the majority of the n-grams, making up 71% (62 out of 87) of the n-gram distribution. Java source code files are all text files and, as inspection of the distribution of common n-grams for Java files reveal, all n-grams are whole English words or fragments of common English words. Further, the n-gram distribution for Java files was larger than the distributions for the CSV, SQL, and TEXT filetypes, which helps to explain how/why these filetypes were commonly mistaken for Java source code file fragments; there were more English n-grams in the Java source code n-gram distribution than the others.

The JPEG filetype also exhibited some interesting false positives. As was discussed earlier, file containers can embed entire files within themselves. The JPEG filetype had false positives coming from PDF, PPS, and SWF files. Each of these three filetypes are file containers. The PDF filetype has been extensively covered within the discussion of this thesis. The PPS filetype is a Microsoft Office powerpoint file that can be viewed by MS PowerPoint or any powerpoint capable viewer. These files, once created, are read-only and can directly embed many other files including JPEG files. The SWF filetype is an Adobe "Small Web Format" file. These files are used to store multimedia content for mobile applications and can be viewed by any Internet browser that has the appropriate plug-in installed. These files store text, audio, and video content and are capable of being interactive. The SWF specification [20] states that JPEG files can be directly embedded within a SWF file. Files from each of the three filetypes were inspected and it was discovered that they did in fact contain JPEG images embedded within the data. For example, a 512-byte file fragment in the test set for this experiment originating from the *govdocs1* file 281940.pps (beginning at decimal byte offset 232) contained a valid JFIF header. This caused the file fragment to be classified as a JPEG rather than its proper classification of PPS. The other two filetype fragments (PDF and SWF) also contained valid JPEG data within their data for this Experiment. The GIF filetype also had similar false positive characteristics, as seen in Table 4.3 in column *"f"*, where file fragments from file containers such as PDF, PPT, PPS, and SWF were classified as GIF file fragments.

Although the overall accuracy for this experiment was quite low, the simple classification method used to classify files produced some interesting results as seen in Table 4.3. Although EPS and PS files are different filetypes, they are closely related. The same can be said for MS

Office Open XML files relating to ZIP files and all purely textual files such as HTML, SQL, TEXT, XML, and TTF having the same data type. While each of the filetypes used in Experiment 1 and Experiment 2 are distinct filetypes and require applications that adhere to the respective file format specifications to read and write to files of these filetypes, many of the filetypes can be said to be in the same class of another filetype or a subset of another distinct filetype. For example, the complete file 281940.pps, discussed in the previous paragraph, was classified as a PPT file in Experiment 1. This is an incorrect classification. However, a PPS file is essentially a read-only PPT file meant to be viewed as a slideshow. In this respect, the classification process was able to make a classification that was at least related to the correct filetype.

The Open XML documents are stored within a ZIP compressed file. Any ZIP capable archive viewer can view/extract the data within any MS Office Open XML file. However, the MS Office suite will not know what to do with a ZIP file that contains non Open XML data. However, Table 4.3 shows the vast majority of DOCX and PPTX file fragments and the majority of the XLSX fragments were classified as ZIP files.

4.4 Experiment 3 – Axelsson's Relabeled Test Set

In order to make a better evaluation of the performance of long n-grams for filetype identification, the Axelsson test set used for Experiments 1 and 2 was relabeled to reflect the proper filetypes of all the files in the set. Experiments 1 and 2 were then re-run in order to see if any improvements to long n-gram filetype identification were made. Since all the JS, ZIP, and JAR files in the original test set were labeled improperly, those filetypes will not be included within the resulting confusion matrices since these types are no longer present within the correctly labeled test set. Further, three files in the original set were renamed as DBASE3 files after the *govdocs1*1 corpus was corrected [21]. I did not have any feature distributions created for this filetype and upon inspection of these three files, discovered that these files contained a large amount of ASCII text. It was decided to run the newly renamed sets through *"feature-comp"* without building a feature distribution for DBASE3 files. These three DBASE3 files and associated fragments were identified as Java source code files/fragments. The reason for this identification is explained in section 4.5.

Table 4.4 shows the confusion matrix of complete file classification on the properly labeled Axelsson set. The accuracy for full file identification decreased from 49% to 48%. However, the average precision increased from 51% to 56% and average recall increased from 48% to

60%. The biggest improvement in precision was the HTML filetype. In Experiment 1, the HTML filetype had a low 42% precision. With the relabeled set, the precision rose to 100%. In the original set, there were only nine HTML files in the set. After the set was relabeled, this number dramatically increased to 57. As was discussed in the beginning if this chapter, many of the improperly labeled files were actually HTML files. This is a result of the *govdocs1* generation from US Government websites where a file was requested from a government server that did not exist. Instead of obtaining the requested file, the webserver generated an HTML error page listing the name of the file that was requested. This HTML page was then saved into the *govdocs1* collection with the name of the requested file. This issue has since been corrected in the *govdocs1.1* corpus [21].

File Type	Classified As																									Files	Recall (%)
	BMP	CSV	DOC	DOCX	EPS	GIF	GZ	HTML	JAVA	JPEG	PDF	PNG	PPS	PPT	PPTX	PS	SQL	SWF	TEXT	TTF	UNK	XBM	XLS	XLSX	XML		
BMP	$\boxed{8}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	100
CSV	0	$\boxed{0}$	1	0	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0
DOC	0	0	$\boxed{2}$	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	5	40
DOCX	0	0	0	$\boxed{8}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	100
EPS	0	0	0	0	$\boxed{10}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	100
GIF	1	0	0	0	0	$\boxed{9}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	90
GZ	0	1	0	0	0	0	$\boxed{9}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	90
HTML	0	0	0	0	0	0	0	$\boxed{7}$	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	57	12
JAVA	0	0	0	0	0	0	0	0	$\boxed{9}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	100
JPEG	0	0	0	0	0	0	0	0	0	$\boxed{10}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	100
PDF	0	0	0	0	0	0	0	0	0	0	$\boxed{10}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	100
PNG	0	0	0	0	0	0	0	0	0	0	0	$\boxed{10}$	0	0	0	0	0	0	0	0	0	0	0	0	0	10	100
PPS	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{0}$	10	0	0	0	0	0	0	0	0	0	0	0	10	0
PPT	0	0	0	0	0	0	0	0	0	1	0	0	0	$\boxed{8}$	0	0	0	0	0	0	0	0	0	0	0	9	88
PPTX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{10}$	0	0	0	0	0	0	0	0	0	0	10	100
PS	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	$\boxed{0}$	0	0	0	0	0	0	0	0	0	5	0
SQL	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	$\boxed{0}$	0	0	0	0	0	0	0	1	6	0
SWF	2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{1}$	0	0	0	0	0	2	0	8	12
TEXT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{0}$	0	0	0	0	0	0	0	0
TTF	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{0}$	0	0	0	0	1	2	0
UNK	0	5	1	0	0	0	0	0	19	0	0	1	3	0	0	0	1	0	0	0	$\boxed{0}$	0	0	0	10	40	0
XBM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{2}$	0	0	0	2	100
XLS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{5}$	0	0	5	100
XLSX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	$\boxed{5}$	0	6	83
XML	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	$\boxed{6}$	7	85
Total Classifications	11	10	4	8	15	9	9	7	92	11	10	11	4	21	10	0	1	1	0	0	0	2	6	7	18	267	
Precision (%)	72	0	50	100	66	100	100	100	9	90	100	90	0	38	100	0	0	100	0	0	0	100	83	71	33		

Table 4.4: Axelsson NEW-EXP-0 File Set – full file classification confusion matrix with correctly labeled files

The PS and EPS files (Adobe PostScript and Encapsulated PostScript files respectively) are essentially the same filetype with the exception that EPS files are meant to be encapsulated within PS files and therefore contain boundary boxes within its syntax and will not include PS "`newpage`" commands. Otherwise, the syntax for both filetypes are identical. In Table 4.4 the precision for PS files was zero, meaning there were zero PS files identified as PS files. As the PS column in Table 4.4 shows, there were also no false positives made, which is still significant since no other non PS files were mistakenly classified as PS files. The EPS recall was 100%, meaning all EPS files were properly identified as EPS files. However, the precision for EPS files was only 66%. Looking at the EPS column in Table 4.4, 10 out of 15 classifications made were correct, the remaining 5 false positives were all five PS files in the test set being misclassified. While these five classifications are incorrect, they are are not completely false due to the fact that PS and EPS files are essentially the same filetype.

Overall, the performance of full file filetype classification for full files can be attributed to the presence of distinguishing long n-grams within the filetype. For example, BMP, PNG, GIF filetypes only contain distinguishing n-grams within the file header, any data after the file header is not structured in a way that is distinct to the filetype. The precision for PNG and GIF files were both 100% and the BMP precision was 72%, which is still a good result for a filetype with minimal distinct n-grams. For all four of these filetypes, the long n-gram approach to filetype identification worked as well as file header identification (with the exception of the BMP precision).

Other filetypes, such as the Open XML Microsoft Office files, performed better for long n-gram analysis than it would have for file header analysis or byte frequency distribution. File header analysis on these filetypes would only be able to reveal these filetypes as being ZIP files, not a wrong classification but still lacking in a more precise identification. Byte frequency analysis (using unigrams or bigrams) would only reveal the data to be high entropy and, depending on the classification method, would only be able to identify these filetypes as ZIP files. However, as Table 4.4 shows for the DOCX, PPTX, and XLSX filetypes, long n-gram analysis have great success at identifying these file not only as Open XML files but was able to correctly identify all DOCX and PPTX files and had a 71% precision for XLSX files. These filetypes, although containing a large amount of high entropy data, contain very distinctive long n-grams that are not part of the file header and allow for easy identification of these filetypes.

Textual files is where long n-gram analysis failed to provide accurate results for specific filetypes. Textual filetypes such as CSV, SQL, and Java Source code do not have distinctive n-grams that can be used for identification. However, Table 4.4 shows textual filetypes all seemed to cluster around the Java source code files. The reason for textual files and file fragments clustering around a single filetype will be discussed further in the following sections. A simple explanation for these results is that rather than identifying files based on distinctive n-grams, these filetypes are being identified based on the textual content of the files.

Table 4.5 shows the same characteristic precision increases for HTML, XLS, and XML filetype classification of file fragments. The same characteristics with respect to fragments coming from embedded files in file containers and related filetypes being classified together are apparent in Table 4.5 as they are in Table 4.3.

File Type	Classified As																									Files	Recall (%)
	BMP	CSV	DOC	DOCX	EPS	GIF	GZ	HTML	JAVA	JPEG	PDF	PNG	PPS	PPT	PPTX	PS	SQL	SWF	TEXT	TTF	UNK	XBM	XLS	XLSX	XML		
BMP	31	15	0	0	0	0	0	0	0	0	0	23	0	0	1	0	0	2	0	0	9	0	3	13	0	97	31
CSV	0	35	0	0	0	0	0	0	85	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	122	28
DOC	62	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	12	0	75	0
DOCX	73	3	0	0	1	0	0	0	0	0	1	4	0	0	0	0	4	0	0	0	0	0	0	34	0	120	0
EPS	0	0	0	0	124	0	0	0	2	0	0	0	0	0	0	24	0	0	0	0	0	0	0	0	0	150	82
GIF	35	0	0	0	0	33	0	0	0	0	0	39	0	0	0	0	7	9	0	0	0	0	3	6	0	132	25
GZ	0	18	0	0	2	18	9	0	0	1	0	8	0	0	0	0	53	15	7	0	0	0	0	13	0	144	6
HTML	0	0	0	0	0	0	0	15	544	0	0	0	31	0	0	0	40	0	0	0	0	0	3	0	59	692	2
JAVA	0	0	23	0	0	0	0	0	104	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	127	81
JPEG	15	6	0	0	0	9	0	0	0	64	0	12	0	0	5	0	13	9	4	0	0	0	0	13	0	150	42
PDF	0	2	0	0	0	14	0	0	0	15	80	0	0	0	0	0	6	0	0	0	0	0	33	0	0	150	53
PNG	0	0	0	0	0	42	0	7	0	0	0	54	0	0	0	0	31	0	0	0	0	0	0	16	0	150	36
PPS	5	2	0	0	0	12	0	0	0	38	2	54	0	0	6	0	10	11	1	0	0	0	1	8	0	150	0
PPT	61	7	0	0	0	8	1	0	0	0	0	31	0	1	0	0	7	0	1	0	0	2	4	12	0	135	0
PPTX	37	4	0	0	0	6	2	0	0	0	0	12	1	0	0	0	3	4	0	0	0	0	0	81	0	150	0
PS	0	0	0	0	27	0	0	0	0	0	0	0	0	0	0	42	0	0	0	0	0	0	0	0	6	75	56
SQL	0	0	0	0	0	0	0	0	32	0	0	0	4	0	0	0	15	0	0	0	0	0	0	0	0	51	29
SWF	15	32	0	0	0	11	0	0	0	15	0	14	0	0	0	0	1	4	1	0	0	0	0	13	0	106	3
TEXT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TTF	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	17	0
UNK	0	31	1	0	0	0	0	0	123	0	0	2	20	0	0	0	45	0	0	0	0	0	40	0	35	297	0
XBM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	0	0	0	7	100
XLS	34	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	30	11	0	75	40
XLSX	64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	15	0	90	16
XML	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	62	65	95
Total Classifications	432	170	24	0	154	153	12	22	893	133	83	253	57	1	12	66	235	54	14	0	9	9	128	247	166	3327	
Precision (%)	7	20	0	0	80	21	75	68	11	48	96	21	0	100	0	63	6	7	0	0	0	77	23	6	37		

Table 4.5: Axelsson NEW-EXP-0 File Set – 512-byte file fragment classification confusion matrix with correctly labeled files

The BMP, GIF, JPEG, and PNG filetypes all had excellent precision and recall rates for full file identification. These filetypes all contain one or two long distinctive n-grams originating from their respective file headers. The remaining n-grams in the feature distributions were mainly bigrams that could easily occur in random data. However, the performance of these filetypes during full file identification indicate that the one or two n-grams coming from file headers were enough to make accurate classifications and detections, indicating the long n-gram approach works as well as plain file header identification for these filetypes. As was expected for these filetypes due to the vast majority of bi-grams in the feature distributions for each filetype, the GIF, JPEG, and PNG precisions and recalls were significantly lower during fragment identification. For each fragment, the identification process only had the abundance of bi-grams available to make the identifications. Interestingly, where file header identification would not work at all since the file header would more than likely not be present or complete, the precision rate for JPEG fragments was 48% and the precision rates for GIF and PNG were both 21%, making the long n-gram identification method somewhat useful even here.

Like the full file identification performance, successful file fragment classification is again attributed to the presence of long n-grams. When they are not present, there is no identification. Again, take for example the BMP, GIF, and PNG filetypes. For full file identification, these filetypes had excellent precision rates. However, these filetypes only contain distinctive n-grams in the file headers. File fragments will not likely contain a full file header. Therefore, filetypes such as BMP, GIF, and PNG will not have any useful long n-grams to identify. Instead, these filetypes will only be able to be identified through the occurrence of bi-grams (which has been shown to contain a relatively small space of possible values making them not ideal for n-gram analysis) and limited tri-grams in the feature distribution for their respective n-gram feature distributions. Table 4.5 shows just this with the low precision and recall rates for these filetypes.

4.5 N-Gram Detection Within Filetypes

Table 4.6 lists some exemplar n-grams from each filetype during two runs of *"featurecomp"* on the test sets used in section 4.4: one run on the 267 properly labeled full file set and the other on the 3327 file fragment set. The column *"Occurrences"* contains two sub-columns. The two sub-columns pertain to the number of individual files and fragments the n-gram occurred at least once in. The n-grams listed are of interest for discussion and to help understand the performance data contained within Tables 4.4 and 4.5. In order to make the information contained in Table 4.6 more concise, all n-grams are displayed in printable ASCII. However, some n-grams contain non printable ASCII byte values. In this case, non printable ASCII byte values are displayed as “.”. In some cases it is necessary to distinguish between a simple “.” and the particular byte value(s) that make up an n-gram. Therefore, certain n-grams contain hex values in parenthesis in order to give the reader a clearer understanding of the n-gram byte values.

		Occurrences	
		n Files	*n* Fragments
File	Summarized N-gram	out of	out of
Type	(hex values in parens)	267	3327
BMP	bm	149	94
	....	74	555
	..(0x01).	63	194
	..(..	43	35
CSV	5555	208	1375
	,5	136	185
	555,	86	143
HTML	ref	133	682
	(0x20)the(0x20)	131	265
	gov	111	339
	></	72	101
	<html	57	573
	<body	57	184
	</html>	57	4
	</body>	57	4
	</head>	55	182
JAVA	55	263	2242
	(0x20)5	252	1649
	5(0x20)	244	1514
	ed	242	1044
	(0x20)(	219	732
	(0x20)(0x20)	213	1226
	ent	187	942
	ion(0x20)	151	542
	(0x20)and(0x20)	119	474
	ble(0x20)	109	342
	public(0x20)	82	553
	(0x20)class	68	224
XML	(0x20)version="5.5"	10	42
	<?xml.version	9	20
DOC	(0x20)55	222	1024
	ation	168	563
	micro	45	2
	word	40	43
	(0x20)document	25	18
DOCX	.[content_types].xml	25	34
	.._rels/.relspk..-..	25	0
	...word/document.xml	8	6
	.word/_rels/document	8	4
PPTX	ppt/viewprops.xmlpk.	10	0
	ppt/slides/_rels/sli	10	53
	,...ppt/slidemasters	10	0
	,...ppt/slidelayouts	10	0
XLSX	xl/_rels/workbook.xm	5	1
	worksheets	11	0
PDF	%%eof	24	9
	5.5.555.555	21	84
	trailer	20	32

	endstream	10	4
	startxref	10	10
	5555555555.55555.f	10	0
	55.5.obj	10	41
	%pdf-5.5	10	2
PS	%%endcomments	16	146
	%%creator:.	16	102
	%!ps-adobe-5.5	16	5
	efont.setfont	13	73
EPS	.5.55	95	473
	color	88	338
	55.55	56	161
	width	84	294
	font	81	219
	showpage	16	15
	moveto	15	127
	(0x20)findfont	11	1
GIF	,....	40	47
	gif55a	15	3
JPEG	(0xFF)(0xC4)	80	100
	(0xFF)(0x00)(0x35)	56	43
	(0x35)(0xFF)(0x00)	53	31
	(0xFF)(0xDA)(0x00)	44	74
	(0xFF)(0xDB)(0x00)	42	77
	......jfif..	39	33
PNG	.png........ihdr..	39	47
	idatx	39	61
	....iend.b‘	39	1

Table 4.6: Occurrences of exemplar n-grams by filetype

The n-gram feature distribution for the BMP filetype only had four n-grams in common to all files in the 20 file learning set. Inspection of the individual BMP n-grams should reveal that these four common BMP n-grams are non-distinct and can easily occur in random data. However, the accuracy for complete BMP file recognition was 100% in section 4.4. For full files, the presence of the BMP file header enabled the long n-gram approach to successfully identify BMP files. When the BMP file header was not present, as in the file fragment test set where 97 BMP file fragments were present, the accuracy for BMP file identification plummeted to 7%: after the 14-byte BMP header, there are no distinguishing characteristics of a BMP file. Therefore, any fragment that originates from somewhere in the middle or the end of a BMP file will not contain any distinct n-grams.

The filetype CSV had poor performance for both complete file fragment identification. The CSV n-gram feature distribution had six total n-grams that were common to all CSV files in the learning set. The exemplars shown in Table 4.6 were the “best” n-grams in the generated feature set and as can be observed, are not at all distinct. The three exemplar n-grams would be found in any textual content that contains for example any four digit number, a comma immediately followed by a single number, or a three digit number immediately followed by a comma. The occurrences of these three n-grams in the counts further support the argument that these n-grams are non-distinct since there were only 10 CSV files in the complete file set and 122 CSV file fragments in the fragment set yet the counts for all three n-grams are significantly higher.

HTML files/fragments carried an excellent precision of 100% for full file identification and a respectable 68% precision for file fragment identification. Although HTML is another purely textual filetype, the relatively distinct nature of HTML tags allow for good HTML file/fragment detection since they do not regularly occur outside of HTML documents. Notice the file counts for the HTML n-grams `</html>`, `</body>`, and `</head>`. There were a total of 57 HTML files in the test set and those n-grams occurred in 57 files out of the entire 267 file test set, indicating that these n-grams only occurred in the HTML files. This helps establish that some of the HTML common n-grams, although purely textual, are distinct enough to classify whole HTML files. However, HTML files and fragments were overwhelmingly classified as Java source code (very low recall rates for both HTML files and fragments in Tables 4.4 and 4.5). The reason for this occurrence is due to the fact that the Java source code n-gram feature distribution had the most textually based bi-grams and tri-grams than any other textual filetype. As seen on Table 4.6 for HTML n-grams, the distinctive n-grams that helped to successfully classify complete HTML files did not occur in all the HTML fragments since those fragment counts were significantly lower than the 578 HTML fragments in the test set. Since the HTML files were taken from US government websites and are purely textual, the abundance of English n-grams and lack of distinctive HTML n-grams caused the HTML file fragments to correlate to the wide range of English n-grams contained in the Java source code feature distribution. Hence the 544 out of 692 HTML fragments being labeled as Java source code fragments in Table 4.5.

The Java files and fragments had very good recalls of 100% and 81% respectively. The precision was very low due to the fact that there were many false positives coming mainly from textual based filetypes. As Table 4.6 shows for the Java source code n-grams, the n-grams are intuitively common among textual files. For example, the bigram $\boxed{\texttt{0x20}}\boxed{\texttt{0x20}}$ is a double space, common to any written English textual content. The n-grams $\boxed{\texttt{e}}\boxed{\texttt{d}}$, $\boxed{\texttt{e}}\boxed{\texttt{n}}\boxed{\texttt{t}}$, and $\boxed{\texttt{i}}\boxed{\texttt{o}}\boxed{\texttt{n}}\boxed{\texttt{0x20}}$ are common English word endings. The n-grams $\boxed{\texttt{p}}\boxed{\texttt{u}}\boxed{\texttt{b}}\boxed{\texttt{l}}\boxed{\texttt{i}}\boxed{\texttt{c}}\boxed{\texttt{0x20}}$ and $\boxed{\texttt{0x20}}\boxed{\texttt{c}}\boxed{\texttt{l}}\boxed{\texttt{a}}\boxed{\texttt{s}}\boxed{\texttt{s}}$, although part of key words in the Java language are again very common English words. Rather than classifying based on characteristics of the Java language, the classifications for the textual filetypes were actually classifying the English content of the Java source code and other textual filetypes. The file and fragment counts for the Java classification help support this argument since there were only nine complete Java source code files and 127 fragments.

The Microsoft Office Open XML files were able to be identified remarkably well even though they are stored within a ZIP file that contains high entropy data. As discussed earlier, ZIP files store the names of files and folders contained within a ZIP archive in plain text. Plain text has been shown so far to be somewhat reliable for filetype identification as long as the n-grams are distinctive, such as in the case of the high precision of HTML filetype identification of complete files. The Open XML format contains distinctive n-grams for the naming scheme that helps to identify a ZIP archive containing Open XML data. The two n-grams `.[content_types].xml` and `.._rels/.relspk..-..` will be found in evry Open XML compliant document file and can be seen by using the command *unzip -l "open_xml_archive_filename"* to view the files and folders inside of a particular Open XML document file.[4] These two n-grams are exemplar n-grams that help to distinguish Open XML files. Further, the Open XML files have even more distinguishing n-grams that correspond to the subset of Open XML files a file belongs to. For example in Table 4.6 for the PPTX, DOCX, and XLSX rows, n-grams reflecting the type of document contained in each Open XML files can be seen. For example the text *"word"* occurring within the DOCX n-grams, the text *"ppt"* occurring in the PPTX n-grams, and the text *"xl"* occurring in the XLSX n-grams. Although the Open XML files contain very specific distinctive n-grams, they are still ZIP archives at a higher level which explains their poor performance during file fragment identification since high entropy data resembles random data.

The PDF filetype experienced very good precision in both full file and file fragment identification. As seen in Table 4.6, the many of the occurrences of the n-grams during complete file identification reflect the number of PDF files in the test set. Further, the fragment counts shown are less than the 150 PDF fragments in the fragment set, implicating that the n-grams in the PDF feature distribution are distinct enough to identify PDF fragments and not be commonly occurring over the entire fragment test set.

The EPS and PS filetypes were closely related to one another for both full file and fragment identification. An EPS file shares the same syntax as a PS file and is meant to be embedded

[4] Since each feature distribution was compared to each file/fragment in both sets, the occurrences for the above n-grams will be equal and referring to the same files/fragments for each feature distribution. This is the reason why common n-grams across the Open XML filetypes in Table 4.6 are only listed under a single row.

within a PS file. In a sense, they could be considered the same types of files. The relation of EPS and PS can be seen in Tables 4.4 and 4.5. For full files identification, 5 out of 5 PS files were incorrectly identified as EPS files. For fragment classification, 27 out of 75 PS fragments were identified as EPS fragments. These are incorrect classifications however it shows that PS files and fragments are again closely related to the EPS filetype. In fact, inspection of the feature distributions for EPS and PS files revealed that they shared many of the same common n-grams. The first three n-grams listed in Table 4.6 for the PS filetype show a file count of 16, possibly indicating that these n-grams were detected within the 5 PS files and 10 EPS files, with an additional occurrence in another filetype.

The GIF, JPEG, and PNG filetype listings in Table 4.6 reveal some interesting results in relation to these types of files being embedded within other filetypes. Take for instance the GIF n-gram gif55a shown in Table 4.6. (Recall that this is a summarized n-gram, so the “55” may in fact be a “87” or an “‘texttt89”. This n-gram, although alphanumeric in content, is a distinctive n-gram since the likelihood of it occurring in random data is very low (due to the length) and low in textual data, unless of course a document was written on the GIF file specification. The occurrences for this n-gram show that it occurred in 15 files. However, only eight files in the full file set were GIF files. This indicates this n-gram was detected within seven other non-GIF files. While this cannot be factually confirmed since the software used to perform the research in this thesis was not set up to record the filenames of the files that this n-gram occurred in, it is still reasonable to assume that some of the files (i.e.,: PDF, PPT, DOC, etc.) in the set contained embedded GIF files since the probability of the n-gram’s occurrence in random data is very low and the likelihood that seven US government files are referring to the GIF file specification is minimal. Further, JPEG and PNG file header n-grams shown in Table 4.6, which are longer than the GIF file header n-gram and thus more distinct, were also seen in a number of files that reasonably reflect the number of file containers in the set.

CHAPTER 5:
Conclusion and Future Work

Filetype identification is really really two related problems: whole file identification and file fragment identification. These problems are different because of the information available to the identification algorithm. For full file identification, algorithms have the luxury of having header and footer information present or at the very least (for filetypes that do not have any header/footers specified) have the full content of a file by which to identify it. File fragments, on the other hand, contain only a fraction (quite possibly a very small fraction) of the data contained within a full file from which it originated and do not have easily recognizable headers or footers. Further, the data contained within a fragment may be indecipherable without any neighboring file fragments or information regarding the offset of where the file fragment originated from the complete file, as is most likely the case with fragments that are compressed. Therefore, file fragment identification must rely on feature identification rather than a file's content or descriptive n-grams (i.e., a file's header).

5.1 Shortcomings of Short N-Grams

N-gram analysis is not a new approach for filetype identification. However, researchers have only used, and continue to use, short n-grams. While some of the research presented in Chapter 2 did have some success, overall the short n-gram approach applied to filetype identification as a general solution is highly inadequate. Short n-grams cannot provide enough of a distinction to be able to fully identify files all the way down to their individual filetypes. For example, some of the approaches in Chapter 2 reported the ability to reliably distinguish between high and low entropy filetypes. While this is helpful to prevent labeling a file or fragment with a filetype associated with the wrong entropy, it does not help to narrow down the appropriate filetype. For instance, a statistical analysis using short n-grams of a text file would be able to distinguish it from a JPEG, ZIP, or DOCX file. However, the same method would have a very difficult time distinguishing between a JPEG, ZIP, or DOCX file since all three would have close to the same entropy level.

5.2 Advantages of Long N-Grams

This thesis explored the use of long n-grams for file classification. As discussed earlier, using long n-grams poses a challenge, as many of the implementation techniques developed for working with short n-grams cannot be readily applied to long n-grams. Rather than tracking the occurrences of n-grams within a multi-dimensional data structure, this thesis shows that it is only important to keep track of commonly occurring n-grams. This can easily be done by building representative file sets of a single filetype and take the intersection of all the files for a particular n-gram size or a range of n-grams sizes. This produces a set of n-grams that are common to all the files in the learning set.

Additional improvements are provided by the use of so-called *summarized n-grams*. Summarized n-grams are created by performing a character transform on the input files prior to the construction of the n-grams. In this case of these thesis, two transformations were used: all capital letters were turned to lower case letters, and all numbers were converted to the number five ("5"). Summarized n-grams significantly improved n-gram detection rates while lowering error rates.

This thesis argues that the length of an n-gram combined with the variability of the individual bytes making up the n-gram is what makes it distinctive: the longer the n-gram is, the less likely it is to occur in random data. But this is only true for n-grams of variable data; it is not true for n-grams that merely repeat a small number of characters (e.g. $\boxed{\texttt{0x00}}\boxed{\texttt{0x01}}\boxed{\texttt{0x00}}\boxed{\texttt{0x01}}\ldots$).

This thesis also noted a potential pitfall of the n-gram approach: when n-grams are based on a filetype's *content* (e.g., English words) rather than *structure* (e.g., HTML tags, JPEG markers, etc..). In such cases n-grams will not be predictive. One way to assure that n-grams are based on structure and not content is to use a variety of different input documents when creating the training set.

It was hypothesized and shown that long distinctive n-grams are present within filetypes that can be used as features for file fragment identification. For some filetypes, there were not any long n-grams that could be used to predict them. Other filetypes had a limited amount of common short n-grams that were not necessarily distinctive (i.e., the segment markers in a JPEG file) but still useful for limited success of fragment identification. Lastly, some of the filetypes tested contained very distinctive long n-grams (augmented by the use of n-gram summarization) that were excellent predictors of filetype for fragments.

5.3 Long N-Grams as a Specialized Approach

For full file identification, distinctive n-grams proved to be successful at predicting a file's type for files that had distinctive n-grams. Table 4.4 lists 14 (out of 25) filetypes that had a precision of 66% or better and were determined to contain distinctive n-grams.

However, as Garfinkel *et al.* stated [9], it is necessary to develop specialized approaches rather than seeking general solutions to file fragment classification. There are filetypes that had no distinctive n-grams (textual based files) and performed poorly on an individual basis but seemed to cluster together to at least identify the content of the files. There were filetypes with limited distinctive n-grams located only in the file header or sparsely located within the file (JPEG, BMP, PNG, GZ, ZIP, etc.). These filetypes performed well for full file identification but had significantly lower scores for fragment identification. Finally, there are filetypes that are filled with long, distinctive n-grams that are excellent candidates for the straightforward long n-gram analysis presented in this thesis (PDF, PS/EPS, HTML).

Long n-gram analysis (or at least the approach presented in this thesis) is not a general solution for file fragment identification, nor is it fully effective for all filetypes in the wild. This thesis has reaffirmed the research of others and shown that there are many file types for which n-grams are not effective in identifying either complete files or file fragments.

However, as a specialized approach, long n-gram analysis is effective at file fragment identification for some popular filetypes. The following section presents some possible directions for future work that could possibly enhance the effectiveness of long n-gram analysis for additional filetype fragments.

5.4 Future Work

5.4.1 More Sophisticated Classification Algorithms

The approach to classification presented in this thesis used a simple scoring algorithm to determine the filetype of a full file or file fragment. A more robust classification algorithm based on the support vector machine, k-nearest neighbor, or decision tree approaches would almost certainly improve the identification results for the filetypes that had low precision ratings.

5.4.2 A Larger Vocabulary of Long N-Grams

Long n-gram analysis only needs to be concerned with keeping track of common n-grams. This thesis focused on n-grams that were common to all files in a set. However, certain filetypes have n-grams that are distinctive yet will not be in every file of that particular type (i.e., optional HTML tags). Therefore, detecting n-grams that are common to a percentage of the files in a set will potentially broaden the range of common n-grams for a filetype. For example, rather than requiring that each n-gram occur in 100% of the files in order to be called common, an n-gram could only be required to be in 75% of the files or another percentage that suits the characteristics of the particular filetype. This way the potential of detecting more distinctive n-grams within a filetype is greater.

After all common n-grams for all the filetypes being tested have been generated, all the n-grams from the various file types can be placed into a single dictionary. Once the dictionary is generated, each unknown file or fragment can then be compared to all the n-grams in the dictionary to produce a vector that will then be fed into a traditional machine learning algorithm.

5.4.3 Embedded Data

File containers can embed files of different types within themselves. Using the long n-gram approach, the different types of files within a container file could be individually identified. For example, a PDF classifier may be able to pick out all PDF files and a JPEG classifier may be able to pick out all JPEG files in a single test set. However, it may be helpful to have any embedded JPEG data located within and PDF files. In this manner, a PDF file is accurately identified and further information regarding the content of the PDF can be revealed.

5.4.4 4096-byte Fragments

Modern computer hard drives are beginning to move away from 512-byte sectors and use 4096-byte (4K) sectors. In a forensics environment, this means that the file fragments recovered from hard drive images will be 4096-bytes in length rather than 512-bytes. For long n-gram analysis, the potential for more precise file fragment identification increases since the length of the fragment will be eight times longer. This may increase the chances of encountering a distinctive n-gram within a 4K file fragment versus a 512-byte fragment that could increase overall accuracy of long n-gram analysis.

REFERENCES

[1] M. B. McDaniel, “An algorithm for content-based automated file type recognition,” Master’s thesis, James Madison University, 2001.

[2] W. Li, K. Wang, S. J. Stolfo, and B. Herzog, “Fileprints: Identifying file types by n-gram analysis,” *Proceedings of the 2005 IEEE Workshop on Information Assurance*, 2005.

[3] M. Karresand and N. Shahmehri, “Oscar—file type identication of binary data in disk clusters and ram pages,” in *Annual Workshop on Digital Forensics and Incident Analysis*. Springer-Verlag, 2006, pp. 85–94.

[4] G. Hall and W. Davis, “Sliding window measurement for file type identification,” Mantech Security and Mission Assurance, 2006.

[5] S. Moody and R. Erbacher, “SADI - statistical analysis for data type identification,” *Third International Workshop on Systematic Approaches to Digital Forensic Engineering*, 2008.

[6] S. Axelsson, “The normalized compression distance as a file fragment classifier,” *The Proceedings of the Tenth Annual DFRWS Conference*, 2010.

[7] S. L. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, “Bringing science to digital forensics with standardized forensic corpora,” in *Proceedings of the 9th Annual Digital Forensic Research Workshop (DFRWS)*, Aug. 2009.

[8] S. Garfinkel and J. Migletz, “New XML-based files: Implications for forensics,” *IEEE Security & Privacy Magazine*, vol. 7, no. 2, Mar. / Apr. 2009.

[9] V. Roussev and S. Garfinkel, “File fragment classification—the case for specialized approaches,” in *Systematic Approaches to Digital Forensics Engineering (IEEE/SADFE 2009)*. Springer, May 2009.

[10] S. Garfinkel, A. Nelson, D. White, and V. Roussev, “Using purpose-built functions and block hashes to enable small block and sub-file forensics,” in *DFRWS 2010*, Portland, OR, 2010.

[11] (2010) TrID homepage. M. Pontello Home Page. [Online]. Available: http://mark0.net/soft-trid-e.html

[12] (2010) Outside in technology. Oracle Corporation. [Online]. Available: http://www.oracle.com/us/technologies/embedded/025613.htm

[13] (2010) UK national archives droid factsheet. UK National Archives. [Online]. Available: http://www.nationalarchives.gov.uk/documents/droid-factsheet-websitefinal.pdf

[14] (2010) GDFR homepage. US National Archives, OCLC, Harvard University. [Online]. Available: www.gdfr.info

[15] G. Conti, S. Bratus, A. Shubina, B. Sangster, R. Ragsdale, M. Supan, A. Lichtenberg, and R. Perez-Alemany, "Automated mapping of large binary objects using primitive fragment type classification," *The Proceedings of the Tenth Annual DFRWS Conference*, 2010.

[16] D. Nadeau, "A survey of named entity recognition," in *Lingvisticae Investigationes*. NRC Institute for Information Technology; National Research Council Canada, 2007.

[17] M. Collins, "Ranking algorithms for named-entity extraction: Boosting and the voted perception," *Proc. Association for Computational Linguistics*, 2002.

[18] (1993) Information technology digital compression and coding of continuous-tone still images requirements and guidelines. [Online]. Available: http://www.w3.org/Graphics/JPEG/itu-t81.pdfl

[19] "JPEG file interchange format," Joint Photographic Experts Group, 1992. [Online]. Available: http://www.jpeg.org/public/jfif.pdf

[20] (2008) Adobe SWF file format specification version 10. Adobe Systems Inc. [Online]. Available: http: //www.adobe.com/content/dam/Adobe/en/devnet/swf/pdf/swf_file_format_spec_v10.pdf

[21] (2010) Announcing govdocs1.1. Digital Corpora Admin. [Online]. Available: http://digitalcorpora.org/archives/143

APPENDIX A:
Experimental File List

048569.bmp	082168.txt	084437.pub	102487.html	104232.pps	120257.gif
130790.ppt	132946.txt	143976.html	145527.pub	155876.pps	164357.pub
174797.html	179487.eps	182614.pptx	183318.csv	183513.html	184216.pdf
186118.bmp	187783.html	191846.pdf	192402.bmp	198075.html	210492.html
212946.xlsx	215737.java	216689.docx	216694.html	221986.html	223624.docx
233200.pps	241163.txt	241343.txt	249421.html	249730.gz	250560.ppt
258068.gz	263566.pptx	268297.html	270700.png	277473.html	277836.csv
281940.pps	289807.gz	295784.csv	295989.bmp	299759.docx	299765.xlsx
307260.ppt	313486.docx	318133.pptx	320417.html	338033.txt	351480.xml
354675.bmp	359397.ps	359739.ps	365256.pub	365624.png	372437.docx
378278.html	380012.java	388207.ppt	388283.xls	391632.gz	392793.html
393001.html	400769.txt	404068.html	404310.html	411102.xlsx	412637.java
412916.html	424036.html	424037.html	424240.bmp	427953.html	427954.html
428151.html	436938.java	446238.html	458821.xls	461510.html	464336.pps
467037.xbm	469082.pdf	469238.html	472113.html	472719.csv	473856.csv
474088.swf	475598.html	475787.pps	476947.csv	478356.csv	483939.html
486901.doc	500968.xlsx	505158.xls	507479.java	509992.txt	511710.csv
519722.gif	528206.xlsx	529625.csv	539314.swf	547335.ppt	553501.swf
559227.html	563294.html	565081.xml	565864.gz	571993.swf	580687.doc
581671.html	589748.swf	589990.gif	593251.ppt	595118.sql	596250.pptx
598299.docx	599295.dbase3	602072.sql	605377.docx	606936.pptx	606985.pub
609019.pptx	610292.swf	614817.pdf	615326.docx	617418.sql	617738.xml
623300.ppt	624005.pdf	624630.csv	629144.html	629593.xml	631855.swf
637614.sql	640685.xml	640953.html	643587.doc	644273.xml	645187.txt
647659.sql	649414.html	650343.dbase3	652358.ps	653042.xml	654985.swf
656598.ttf	656605.eps	656869.html	657039.txt	657428.html	658291.html
658441.html	658475.html	658611.html	658916.html	659230.html	659719.txt
660568.txt	661130.txt	661635.ttf	661649.txt	661821.txt	662201.txt
662373.txt	662566.txt	663105.txt	663272.html	663635.ps	663823.txt
664511.txt	665018.txt	665201.txt	666048.txt	666383.txt	669189.txt
670890.jpg	671354.eps	672116.jpg	675927.pub	680958.eps	687675.doc
690531.html	694506.html	694534.gz	694669.eps	705948.png	710340.jpg
711872.html	712465.pps	712648.html	712928.java	714314.pdf	714859.html
716691.html	717834.ppt	719492.html	720718.png	729601.pptx	739365.jpg
749043.eps	756794.pdf	759867.txt	764163.pdf	785243.eps	808841.rtf
811459.bmp	817954.xls	819781.eps	819840.gz	821535.png	823904.gif
827462.jpg	827478.gif	836334.java	842644.dbase3	848990.xlsx	858257.pdf
862652.png	868217.eps	874196.bmp	875216.gif	880064.doc	881722.sql
890237.gz	901341.xls	902126.txt	902129.html	905600.pdf	907395.jpg
911726.html	912721.txt	912800.gif	919144.gif	920125.png	920697.png
920706.png	921469.gz	922536.java	923701.eps	924835.gif	927344.pps
927624.pps	931077.txt	934809.java	937277.gz	942196.png	942817.ps
945965.jpg	948335.ppt	949268.jpg	956747.jpg	956880.gif	977633.html
978133.pptx	978134.pptx	985704.html	987041.jpg	988313.pps	988932.txt
994170.html	994278.xbm	996017.pptx			

Table A.1: List of files used for Experiment 3. For Experiments 1 and 2, the filenames are the same with the exception of the file extensions for the reasons discussed in the beginning of Chapter 4

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Experimental File Fragment List

Table B.1 shows all of the file fragments used for Experiments 2 and 3 in this thesis. Notice that some byte offsets are sequential to one another. This means that some of the file fragments utilized had overlapping 512-byte fragments.

Filename	Fragment Byte Offsets
048569.bmp	00000, 00001, 00002, 00003, 00004, 00005
082168.txt	00003, 00004, 00008, 00009, 00010, 00018, 00019, 00032 00040, 00045, 00070, 00073, 00082, 00108, 00112
084437.pub	00000
102487.html	00000, 00002, 00005, 00006, 00007, 00009, 00010, 00011 00013, 00014, 00016, 00018, 00019, 00020, 00022
104232.pps	00293, 00345, 00977, 01447, 01659, 01806, 02501, 02560 02782, 03111, 03771, 03962, 04311, 04904, 05004
120257.gif	00001, 00003, 00004, 00007, 00015, 00017, 00024, 00027 00028, 00039, 00045, 00054, 00055, 00064, 00074
130790.ppt	00020, 00039, 00062, 00073, 00074, 00075, 00099, 00114 00149, 00201, 00226, 00253, 00280, 00323, 00328
132946.txt	00000, 00001, 00002, 00003, 00004, 00005
143976.html	00053, 00062, 00070, 00072, 00102, 00110, 00144, 00170 00187, 00215, 00220, 00223, 00226, 00229, 00247
145527.pub	00000
155876.pps	00027, 00080, 00109, 00119, 00128, 00547, 00610, 00625 00684, 00711, 00755, 00858, 00952, 00976, 01024
164357.pub	00000
174797.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012, 00013
179487.eps	00009, 00024, 00097, 00099, 00106, 00139, 00196, 00346 00396, 00461, 00478, 00483, 00549, 00561, 00579
182614.pptx	00436, 00570, 00671, 00797, 00818, 00831, 00927, 01000 01815, 02131, 02510, 02618, 03709, 03727, 03947
183318.csv	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007
183513.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
184216.pdf	00008, 00046, 00069, 00075, 00081, 00120, 00169, 00175 00188, 00191, 00197, 00201, 00215, 00219, 00225
186118.bmp	00000
187783.html	00007, 00016, 00017, 00023, 00028, 00034, 00035, 00036 00041, 00042, 00045, 00048, 00049, 00051, 00056
191846.pdf	00033, 00052, 00108, 00206, 00274, 00372, 00466, 00568 00679, 00781, 00847, 01059, 01210, 01311, 01378

192402.bmp	00011, 00024, 00067, 00076, 00172, 00189, 00200, 00204 00261, 00279, 00400, 00459, 00471, 00524, 00540
198075.html	00000, 00001, 00002, 00003, 00007, 00008, 00012, 00013 00014, 00016, 00017, 00018, 00020, 00021, 00023
210492.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00009, 00010, 00011, 00012, 00013, 00014, 00015
212946.xlsx	00014, 00031, 00044, 00269, 00343, 00500, 00551, 00637 00695, 00838, 00882, 00934, 00947, 01131, 01278
215737.java	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012, 00013, 00014
216689.docx	00006, 00011, 00022, 00033, 00058, 00071, 00075, 00091 00092, 00093, 00098, 00103, 00118, 00119, 00120
216694.html	00000, 00004, 00006, 00007, 00008, 00009, 00011, 00012 00014, 00016, 00017, 00022, 00024, 00025, 00032
221986.html	00000
223624.docx	00171, 00289, 00652, 00752, 01140, 01619, 01858, 02008 02443, 02486, 02503, 02645, 02896, 03168, 03468
233200.pps	00010, 00068, 00245, 00281, 00334, 00343, 00402, 00445 00513, 00548, 00562, 00580, 00614, 00615, 00730
241163.txt	00001, 00006, 00011, 00012, 00014, 00016, 00021, 00022 00038, 00051, 00061, 00073, 00075, 00095, 00099
241343.txt	00002, 00018, 00030, 00039, 00043, 00066, 00075, 00087 00091, 00093, 00096, 00100, 00111, 00121, 00124
249421.html	00060, 00253, 00456, 00502, 00715, 00718, 00986, 01108 01154, 01300, 01327, 01329, 01397, 01651, 01720
249730.gz	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012, 00013, 00014
250560.ppt	00074, 00087, 00103, 00117, 00133, 00134, 00147, 00158 00160, 00173, 00200, 00202, 00235, 00315, 00333
258068.gz	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
263566.pptx	00222, 00422, 00556, 00583, 00633, 00650, 01601, 01633 01693, 01743, 01916, 01990, 02151, 02419, 02586
268297.html	00001, 00008, 00011, 00015, 00025, 00028, 00031, 00036 00037, 00038, 00043, 00044, 00050, 00055, 00057
270700.png	00035, 00077, 00163, 00167, 00190, 00241, 00306, 00323 00326, 00377, 00405, 00420, 00437, 00452, 00496
277473.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012
277836.csv	00017, 00021, 00034, 00037, 00078, 00085, 00119, 00138 00145, 00212, 00225, 00236, 00261, 00314, 00330
281940.pps	00232, 00239, 00274, 00400, 00472, 00782, 01068, 01172 01299, 01447, 01457, 01507, 01650, 01777, 01887
289807.gz	00001, 00002, 00003, 00005, 00007, 00008, 00009, 00011 00013, 00017, 00018, 00019, 00021, 00029, 00030
295784.csv	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011
295989.bmp	00049, 00081, 00082, 00159, 00180, 00210, 00213, 00413

	00426, 00672, 00683, 00808, 00890, 00904, 00916
299759.docx	00014, 00022, 00024, 00037, 00042, 00045, 00051, 00064 00065, 00066, 00072, 00078, 00082, 00085, 00102
299765.xlsx	00000, 00003, 00008, 00009, 00011, 00013, 00020, 00021 00026, 00027, 00028, 00029, 00030, 00033, 00035
307260.ppt	02508, 04046, 06487, 08466, 10780, 11464, 12497, 14949 16508, 17284, 20934, 22097, 27057, 27772, 28361
313486.docx	00134, 01723, 01879, 04072, 04414, 05447, 05655, 09518 10913, 11825, 14169, 14545, 15000, 16043, 16412
318133.pptx	00124, 01013, 01048, 01723, 02345, 02497, 04001, 04442 05711, 05777, 05974, 06165, 06862, 07320, 07663
320417.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
338033.txt	00000, 00003, 00004, 00005, 00006, 00013, 00014, 00015 00016, 00017, 00020, 00021, 00022, 00026, 00027
351480.xml	00000, 00005, 00006, 00009, 00010, 00013, 00014, 00015 00017, 00021, 00022, 00025, 00026, 00027, 00028
354675.bmp	00002, 00005, 00016, 00021, 00028, 00033, 00037, 00064 00068, 00081, 00098, 00109, 00110, 00111, 00115
359397.ps	00090, 00126, 00297, 00490, 00568, 00681, 00819, 01492 01681, 01757, 01849, 01905, 01923, 01926, 02373
359739.ps	00007, 00044, 00078, 00127, 00499, 00580, 00587, 00601 00638, 00682, 00761, 00763, 00832, 00834, 00840
365256.pub	00000
365624.png	00004, 00006, 00011, 00019, 00022, 00024, 00026, 00053 00059, 00075, 00081, 00091, 00097, 00099, 00127
372437.docx	00367, 00409, 00552, 00557, 00608, 00712, 00718, 00828 00893, 00933, 00973, 01019, 01067, 01137, 01331
378278.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011
380012.java	00000, 00001, 00002, 00003, 00005, 00006, 00007, 00010 00011, 00012, 00013, 00014, 00015, 00016, 00017
388207.ppt	00007, 00017, 00041, 00059, 00073, 00088, 00096, 00101 00181, 00205, 00212, 00218, 00225, 00228, 00231
388283.xls	00001, 00006, 00035, 00045, 00061, 00071, 00075, 00085 00089, 00092, 00103, 00105, 00110, 00113, 00137
391632.gz	00120, 00259, 00285, 00330, 00422, 00494, 00668, 00747 01039, 01044, 01062, 01074, 01203, 01267, 01446
392793.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
393001.html	00000, 00001, 00002, 00003, 00004, 00005, 00007, 00008 00009, 00010, 00011, 00012, 00013, 00015, 00016
400769.txt	00000, 00001, 00003, 00004, 00006, 00011, 00016, 00017 00021, 00022, 00024, 00025, 00027, 00031, 00035
404068.html	00002, 00004, 00006, 00009, 00011, 00013, 00021, 00023 00025, 00032, 00033, 00034, 00036, 00037, 00041
404310.html	00000, 00001, 00002, 00003, 00004, 00006, 00007, 00008 00009, 00010, 00011, 00012, 00013, 00014, 00015

411102.xlsx	00003, 00008, 00015, 00025, 00062, 00074, 00076, 00081 00104, 00119, 00124, 00129, 00140, 00147, 00155
412637.java	00001, 00007, 00008, 00013, 00015, 00017, 00018, 00020 00022, 00024, 00029, 00034, 00040, 00042, 00044
412916.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
424036.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
424037.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
424240.bmp	00165, 00371, 00975, 01083, 01442, 01473, 01811, 02163 02165, 02601, 02674, 02725, 02741, 02857, 02862
427953.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011
427954.html	00000, 00001, 00002, 00003, 00004, 00007, 00008, 00009 00011, 00012, 00015, 00016, 00017, 00019, 00022
428151.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012
436938.java	00002, 00004, 00005, 00010, 00014, 00015, 00017, 00019 00020, 00026, 00029, 00030, 00031, 00033, 00039
446238.html	00001, 00003, 00007, 00010, 00014, 00015, 00016, 00017 00022, 00025, 00034, 00045, 00046, 00052, 00065
458821.xls	00035, 00043, 00050, 00066, 00074, 00075, 00125, 00130 00139, 00151, 00152, 00191, 00194, 00204, 00206
461510.html	00000, 00001, 00002, 00003, 00004, 00006, 00011, 00014 00015, 00017, 00018, 00019, 00020, 00021, 00023
464336.pps	00176, 01611, 01748, 02695, 02834, 03321, 03676, 03681 03988, 04582, 05472, 05558, 06332, 06441, 07111
467037.xbm	00000
469082.pdf	00008, 00012, 00013, 00026, 00033, 00037, 00040, 00041 00047, 00054, 00060, 00065, 00074, 00088, 00091
469238.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009
472113.html	00043, 00057, 00070, 00108, 00116, 00130, 00135, 00138 00140, 00203, 00204, 00206, 00233, 00235, 00240
472719.csv	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012, 00013
473856.csv	00009, 00010, 00017, 00020, 00025, 00028, 00034, 00053 00060, 00061, 00067, 00085, 00091, 00125, 00141
474088.swf	00000, 00003, 00009, 00010, 00011, 00012, 00013, 00014 00018, 00019, 00023, 00024, 00025, 00026, 00027
475598.html	00000, 00010, 00011, 00014, 00018, 00019, 00020, 00021 00022, 00026, 00027, 00028, 00029, 00030, 00037
475787.pps	00039, 00044, 00046, 00076, 00086, 00090, 00121, 00133 00194, 00217, 00232, 00245, 00310, 00363, 00470
476947.csv	00000, 00001, 00002, 00003, 00004, 00005, 00006
478356.csv	00000, 00001, 00002, 00003, 00004, 00005
483939.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007

	00008
486901.doc	00000, 00002, 00003, 00007, 00008, 00010, 00016, 00020 00021, 00024, 00025, 00026, 00027, 00030, 00032
500968.xlsx	00000, 00039, 00045, 00048, 00081, 00087, 00093, 00095 00096, 00105, 00111, 00127, 00131, 00132, 00142
505158.xls	00005, 00008, 00009, 00013, 00014, 00015, 00022, 00028 00034, 00041, 00044, 00047, 00054, 00063, 00065
507479.java	00000, 00002, 00003, 00005, 00007, 00008, 00010, 00015 00019, 00021, 00032, 00033, 00034, 00040, 00042
509992.txt	00046, 00096, 00156, 00183, 00196, 00201, 00228, 00230 00232, 00246, 00275, 00282, 00283, 00298, 00307
511710.csv	00000, 00001, 00003, 00004, 00005, 00006, 00007, 00008 00009, 00011, 00012, 00013, 00016, 00017, 00021
519722.gif	00000, 00002, 00006, 00008, 00013, 00015, 00020, 00021 00025, 00028, 00031, 00038, 00039, 00041, 00045
528206.xlsx	00000, 00001, 00002, 00004, 00008, 00009, 00010, 00012 00015, 00016, 00018, 00020, 00022, 00024, 00025
529625.csv	00000, 00001, 00002, 00003, 00006, 00007, 00008, 00010 00011, 00012, 00013, 00015, 00016, 00018, 00019
539314.swf	00000, 00001, 00003, 00005, 00006, 00007, 00008, 00010 00012, 00013, 00014, 00017, 00018, 00020, 00021
547335.ppt	01657, 01792, 02784, 03029, 04014, 04829, 05849, 06131 07078, 07382, 08985, 12147, 12312, 13918, 14310
553501.swf	00035, 00072, 00084, 00098, 00126, 00132, 00148, 00166 00177, 00181, 00193, 00198, 00217, 00228, 00275
559227.html	00000
563294.html	00000
565081.xml	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010
565864.gz	00014, 00029, 00055, 00124, 00132, 00196, 00213, 00218 00237, 00292, 00356, 00360, 00395, 00402, 00428
571993.swf	00001, 00007, 00026, 00031, 00032, 00037, 00038, 00042 00047, 00057, 00058, 00059, 00060, 00064, 00068
580687.doc	00012, 00047, 00050, 00066, 00077, 00079, 00087, 00096 00109, 00112, 00113, 00120, 00122, 00124, 00134
581671.html	00003, 00004, 00005, 00006, 00007, 00008, 00010, 00012 00013, 00014, 00015, 00019, 00021, 00022, 00026
589748.swf	00009, 00042, 00045, 00059, 00071, 00088, 00095, 00100 00150, 00151, 00173, 00182, 00187, 00204, 00216
589990.gif	00000, 00001, 00002, 00003
593251.ppt	00096, 00151, 00155, 00245, 00471, 00533, 00595, 00620 00626, 00634, 00646, 00699, 00822, 00835, 00941
595118.sql	00000, 00001, 00002, 00003, 00004
596250.pptx	02685, 03153, 04977, 06228, 06323, 07414, 08120, 08370 09652, 10332, 11807, 11821, 11875, 13027, 13563
598299.docx	00024, 00068, 00090, 00107, 00108, 00113, 00129, 00137 00147, 00148, 00150, 00159, 00160, 00172, 00193
599295.dbase3	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007

	00008, 00009, 00010, 00011, 00012, 00013
602072.sql	00000, 00002, 00003, 00006, 00007, 00009, 00010, 00011 00012, 00013, 00014, 00015, 00016, 00017, 00018
605377.docx	00001, 00002, 00004, 00005, 00006, 00007, 00008, 00009 00010, 00011, 00012, 00014, 00015, 00016, 00019
606936.pptx	00378, 00465, 00516, 00637, 00713, 00976, 01404, 01762 01856, 02169, 02236, 02257, 02340, 02701, 02783
606985.pub	00000, 00001
609019.pptx	00173, 00381, 00611, 01106, 01380, 01838, 01958, 04607 04757, 04907, 05139, 05351, 05488, 06202, 06753
610292.swf	00645, 00718, 00918, 01018, 01340, 01617, 01898, 02179 02272, 02463, 03605, 03740, 04005, 04119, 04177
614817.pdf	00052, 00083, 00200, 00276, 00553, 00656, 00720, 00925 01303, 01379, 01520, 01566, 01640, 01656, 01685
615326.docx	00003, 00006, 00007, 00008, 00009, 00012, 00013, 00021 00024, 00025, 00027, 00029, 00032, 00035, 00037
617418.sql	00000, 00001, 00002, 00003
617738.xml	00000, 00001, 00002, 00003, 00004, 00005
623300.ppt	00131, 00336, 00573, 00678, 01517, 01529, 02339, 02435 03161, 03345, 03739, 04074, 04236, 04904, 05189
624005.pdf	00009, 00022, 00030, 00032, 00033, 00041, 00044, 00046 00070, 00073, 00088, 00099, 00105, 00107, 00114
624630.csv	00000, 00001, 00003, 00005, 00007, 00008, 00010, 00011 00012, 00013, 00016, 00017, 00018, 00019, 00020
629144.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
629593.xml	00000, 00001, 00002
631855.swf	00115, 00527, 00971, 01137, 01293, 01795, 02188, 02307 02367, 02390, 02415, 02439, 02675, 02781, 02959
637614.sql	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010
640685.xml	00001, 00002, 00004, 00006, 00008, 00012, 00013, 00014 00015, 00019, 00020, 00021, 00022, 00023, 00025
640953.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
643587.doc	00052, 00211, 00280, 00386, 00580, 00661, 00742, 00751 00796, 00889, 00909, 00928, 00932, 01102, 01126
644273.xml	00000, 00001, 00002, 00003, 00004, 00005
645187.txt	00003, 00004, 00006, 00008, 00016, 00017, 00018, 00023 00025, 00034, 00036, 00038, 00039, 00044, 00047
647659.sql	00000
649414.html	00013, 00030, 00031, 00035, 00041, 00042, 00043, 00050 00071, 00081, 00101, 00126, 00128, 00153, 00176
650343.dbase3	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012
652358.ps	00000, 00040, 00056, 00066, 00086, 00088, 00097, 00115 00143, 00192, 00245, 00309, 00398, 00501, 00506
653042.xml	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007

	00008
654985.swf	00000
656598.ttf	00000, 00001
656605.eps	00005, 00006, 00026, 00041, 00051, 00059, 00060, 00079 00084, 00109, 00110, 00117, 00121, 00124, 00131
656869.html	00000
657039.txt	00000, 00001, 00002, 00003, 00004
657428.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012, 00013, 00014
658291.html	00000, 00002, 00005, 00006, 00007, 00008, 00009, 00010 00011, 00012, 00013, 00014, 00015, 00016, 00017
658441.html	00000, 00002, 00003, 00006, 00007, 00008, 00009, 00010 00011, 00013, 00014, 00016, 00017, 00018, 00020
658475.html	00000, 00002, 00003, 00004, 00005, 00009, 00013, 00014 00016, 00021, 00022, 00023, 00024, 00025, 00029
658611.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00011 00013, 00014, 00015, 00016, 00017, 00018, 00019
658916.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
659230.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009
659719.txt	00000, 00001, 00002, 00003, 00004
660568.txt	00000, 00001, 00002, 00003, 00004
661130.txt	00000, 00001, 00002, 00003, 00004
661635.ttf	00016, 00030, 00038, 00045, 00069, 00078, 00085, 00096 00109, 00126, 00138, 00144, 00168, 00181, 00211
661649.txt	00000, 00001, 00002, 00003, 00004
661821.txt	00000, 00001, 00002, 00003, 00004, 00005, 00006
662201.txt	00000, 00001, 00002, 00003, 00004
662373.txt	00002, 00004, 00005, 00006, 00007, 00008, 00010, 00011 00013, 00014, 00017, 00018, 00020, 00021, 00023
662566.txt	00000, 00001, 00002, 00003, 00004
663105.txt	00000, 00001, 00002, 00003, 00004
663272.html	00000, 00001, 00002, 00004, 00005, 00008, 00010, 00011 00012, 00014, 00016, 00018, 00020, 00021, 00022
663635.ps	00165, 00180, 00186, 00218, 00247, 00286, 00382, 00429 00447, 00493, 00552, 00605, 00608, 00660, 00663
663823.txt	00000, 00001, 00002, 00003, 00004
664511.txt	00000, 00001, 00002, 00003, 00004
665018.txt	00000, 00001
665201.txt	00000, 00001, 00002, 00003, 00004
666048.txt	00000, 00001, 00002, 00003, 00004
666383.txt	00000, 00001, 00002, 00003, 00004
669189.txt	00003, 00004, 00005, 00006, 00007, 00010, 00011, 00013 00014, 00016, 00017, 00019, 00020, 00022, 00023
670890.jpg	00047, 00130, 00174, 00176, 00253, 00285, 00306, 00316 00326, 00328, 00333, 00377, 00518, 00541, 00568
671354.eps	00001, 00003, 00005, 00007, 00016, 00017, 00021, 00031

	00035, 00049, 00062, 00066, 00079, 00082, 00088
672116.jpg	00005, 00007, 00030, 00038, 00062, 00064, 00079, 00100 00114, 00124, 00128, 00130, 00132, 00135, 00144
675927.pub	00000
680958.eps	00081, 00086, 00093, 00104, 00114, 00337, 00469, 00481 00522, 00527, 00585, 00613, 00720, 00825, 01048
687675.doc	00001, 00003, 00011, 00012, 00023, 00027, 00028, 00033 00035, 00038, 00051, 00053, 00056, 00057, 00065
690531.html	00000, 00017, 00046, 00071, 00089, 00098, 00117, 00122 00128, 00132, 00147, 00164, 00166, 00172, 00180
694506.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008
694534.gz	00021, 00036, 00064, 00092, 00126, 00146, 00176, 00186 00235, 00247, 00284, 00313, 00342, 00358, 00363
694669.eps	00163, 00893, 01433, 01512, 01649, 01779, 02034, 02635 03103, 04133, 04301, 04309, 04467, 05279, 05669
705948.png	00014, 00017, 00028, 00030, 00034, 00047, 00069, 00096 00099, 00119, 00126, 00163, 00172, 00181, 00193
710340.jpg	00000, 00001, 00002, 00003, 00004, 00007, 00008, 00009 00010, 00011, 00012, 00014, 00015, 00016, 00017
711872.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012
712465.pps	00422, 00438, 00704, 01025, 01540, 01647, 02028, 03271 03661, 05827, 05842, 08222, 08666, 09033, 09508
712648.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012
712928.java	00000, 00002, 00006, 00016, 00027, 00033, 00039, 00041 00052, 00069, 00092, 00103, 00123, 00124, 00129
714314.pdf	00008, 00011, 00012, 00013, 00017, 00019, 00023, 00024 00042, 00045, 00046, 00050, 00051, 00056, 00064
714859.html	00000, 00001, 00002, 00003, 00004, 00006, 00007, 00008 00009, 00010, 00011, 00012, 00013, 00015, 00016
716691.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010
717834.ppt	00000, 00010, 00013, 00023, 00025, 00027, 00031, 00046 00060, 00063, 00066, 00068, 00070, 00091, 00103
719492.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012
720718.png	00008, 00013, 00033, 00046, 00052, 00070, 00084, 00102 00108, 00113, 00123, 00136, 00143, 00159, 00167
729601.pptx	00768, 01051, 01083, 01922, 02636, 02648, 02864, 03646 03741, 04524, 05037, 06102, 06320, 06798, 06998
739365.jpg	00015, 00020, 00021, 00061, 00095, 00114, 00130, 00143 00187, 00204, 00233, 00271, 00273, 00332, 00333
749043.eps	00004, 00005, 00008, 00011, 00022, 00023, 00028, 00030 00032, 00034, 00036, 00038, 00039, 00043, 00045
756794.pdf	00031, 00137, 00162, 00171, 00197, 00202, 00209, 00264 00280, 00298, 00374, 00406, 00432, 00455, 00469

759867.txt	00004, 00006, 00018, 00029, 00042, 00044, 00045, 00046 00075, 00081, 00085, 00089, 00091, 00101, 00102
764163.pdf	00000, 00003, 00007, 00009, 00012, 00015, 00018, 00019 00025, 00027, 00031, 00033, 00037, 00039, 00040
785243.eps	00006, 00012, 00014, 00015, 00016, 00019, 00020, 00021 00028, 00039, 00044, 00049, 00050, 00063, 00066
808841.rtf	00003, 00053, 00101, 00120, 00159, 00179, 00193, 00216 00222, 00256, 00261, 00269, 00281, 00287, 00309
811459.bmp	00001, 00002, 00004, 00005, 00008, 00009, 00016, 00018 00019, 00023, 00026, 00029, 00030, 00031, 00033
817954.xls	00040, 00043, 00045, 00056, 00066, 00081, 00089, 00101 00134, 00142, 00169, 00172, 00175, 00199, 00201
819781.eps	00015, 00019, 00021, 00025, 00028, 00035, 00043, 00055 00062, 00064, 00075, 00082, 00084, 00092, 00098
819840.gz	00036, 00108, 00111, 00135, 00164, 00284, 00288, 00296 00305, 00330, 00411, 00412, 00449, 00469, 00491
821535.png	00004, 00008, 00009, 00010, 00011, 00012, 00014, 00015 00019, 00020, 00024, 00025, 00028, 00029, 00030
823904.gif	00011, 00014, 00024, 00026, 00031, 00032, 00043, 00062 00065, 00068, 00075, 00076, 00089, 00099, 00105
827462.jpg	00028, 00055, 00169, 00585, 00613, 00811, 00824, 01300 01342, 01345, 01489, 01568, 01583, 02159, 02189
827478.gif	00007, 00009, 00014, 00030, 00031, 00033, 00048, 00055 00083, 00089, 00100, 00115, 00126, 00133, 00164
836334.java	00017, 00025, 00027, 00028, 00042, 00048, 00059, 00063 00066, 00076, 00079, 00080, 00081, 00085, 00086
842644.dbase3	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007
848990.xlsx	00009, 00011, 00012, 00013, 00014, 00016, 00021, 00026 00033, 00035, 00036, 00037, 00040, 00041, 00044
858257.pdf	00006, 00039, 00048, 00065, 00070, 00082, 00104, 00124 00125, 00130, 00142, 00145, 00150, 00156, 00166
862652.png	00027, 00040, 00041, 00043, 00051, 00059, 00061, 00074 00075, 00083, 00088, 00101, 00132, 00145, 00146
868217.eps	00006, 00007, 00013, 00019, 00023, 00029, 00037, 00039 00040, 00045, 00046, 00049, 00058, 00061, 00074
874196.bmp	00064, 00093, 00488, 00903, 00946, 01465, 01576, 01598 01621, 01749, 01793, 02003, 02280, 02290, 02778
875216.gif	00008, 00013, 00019, 00040, 00048, 00052, 00054, 00058 00085, 00090, 00093, 00095, 00098, 00099, 00118
880064.doc	00000, 00001, 00003, 00004, 00005, 00009, 00010, 00013 00023, 00032, 00036, 00037, 00039, 00041, 00042
881722.sql	00000, 00004, 00006, 00007, 00008, 00009, 00010, 00011 00012, 00013, 00015, 00017, 00018, 00019, 00020
890237.gz	00022, 00101, 00110, 00119, 00220, 00280, 00391, 00429 00475, 00494, 00504, 00514, 00526, 00548, 00627
901341.xls	00001, 00002, 00006, 00011, 00012, 00013, 00014, 00019 00022, 00025, 00029, 00031, 00032, 00037, 00038

902126.txt	00000, 00001
902129.html	00010, 00015, 00021, 00030, 00033, 00037, 00048, 00049 00052, 00056, 00086, 00091, 00105, 00130, 00131
905600.pdf	00000, 00001, 00003, 00004, 00005, 00006, 00007, 00008 00010, 00011, 00013, 00014, 00015, 00017, 00019
907395.jpg	00000, 00007, 00008, 00011, 00013, 00014, 00016, 00021 00034, 00039, 00043, 00046, 00048, 00050, 00057
911726.html	00063, 00105, 00159, 00165, 00177, 00179, 00199, 00208 00213, 00321, 00344, 00367, 00424, 00428, 00536
912721.txt	00000, 00001
912800.gif	00004, 00006, 00012, 00014, 00016, 00017, 00024, 00026 00031, 00037, 00041, 00042, 00049, 00052, 00054
919144.gif	00006, 00007, 00014, 00021, 00030, 00032, 00061, 00062 00065, 00075, 00076, 00092, 00095, 00097, 00104
920125.png	00013, 00018, 00027, 00085, 00134, 00157, 00211, 00256 00259, 00290, 00334, 00386, 00432, 00487, 00541
920697.png	00003, 00148, 00208, 00228, 00230, 00258, 00263, 00355 00366, 00403, 00405, 00439, 00464, 00497, 00573
920706.png	00052, 00079, 00118, 00129, 00136, 00182, 00216, 00241 00249, 00269, 00275, 00409, 00428, 00535, 00620
921469.gz	00018, 00056, 00069, 00080, 00092, 00112, 00115, 00118 00119, 00120, 00141, 00142, 00149, 00163, 00169
922536.java	00000, 00001, 00002, 00003, 00004, 00005, 00006
923701.eps	00005, 00006, 00007, 00008, 00009, 00010, 00019, 00023 00024, 00026, 00027, 00029, 00030, 00031, 00034
924835.gif	00046, 00070, 00121, 00220, 00246, 00396, 00410, 00467 00671, 00851, 00926, 01072, 01124, 01133, 01573
927344.pps	00317, 00713, 00894, 00896, 00986, 01032, 01179, 01597 01611, 01881, 01998, 02484, 02535, 02706, 02790
927624.pps	00244, 00311, 00378, 00589, 00681, 00790, 01092, 01237 01347, 01412, 01592, 01666, 01692, 01723, 01911
931077.txt	00000, 00001, 00002
934809.java	00000, 00003, 00005, 00007, 00009, 00011, 00014, 00017 00018, 00021, 00024, 00025, 00026, 00028, 00029
937277.gz	00050, 00055, 00435, 00762, 00765, 01008, 01009, 01115 01136, 01164, 01188, 01399, 01635, 01822, 01868
942196.png	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011, 00012, 00013, 00014
942817.ps	00006, 00045, 00062, 00149, 00164, 00202, 00233, 00273 00310, 00377, 00406, 00494, 00524, 00632, 00744
945965.jpg	00152, 00181, 00724, 00731, 00830, 00873, 00910, 01266 01382, 01400, 01659, 01777, 01909, 02150, 02203
948335.ppt	01388, 02199, 03556, 03761, 03886, 04947, 05043, 06238 10615, 10618, 11389, 11486, 12119, 14705, 14865
949268.jpg	00030, 00056, 00143, 00332, 00573, 00645, 00700, 00716 00785, 00797, 01161, 01164, 01167, 01226, 01324
956747.jpg	00002, 00007, 00018, 00046, 00072, 00074, 00078, 00084

	00096, 00097, 00101, 00103, 00106, 00114, 00133
956880.gif	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007
977633.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007 00008, 00009, 00010, 00011
978133.pptx	00127, 00517, 00949, 01197, 01472, 01476, 01514, 03395 03501, 03992, 04443, 05688, 07090, 07205, 07660
978134.pptx	00061, 00062, 00069, 00091, 00120, 00122, 00128, 00165 00210, 00227, 00297, 00306, 00315, 00317, 00326
985704.html	00006, 00009, 00015, 00023, 00024, 00035, 00037, 00044 00047, 00056, 00057, 00060, 00061, 00062, 00064
987041.jpg	00025, 00090, 00428, 00460, 00540, 00680, 00750, 00851 00948, 01014, 01083, 01153, 01163, 01276, 01407
988313.pps	00305, 01417, 01568, 02682, 03414, 03554, 03599, 04314 04640, 04675, 04745, 05075, 05223, 05535, 05598
988932.txt	00000, 00001, 00002
994170.html	00000, 00001, 00002, 00003, 00004, 00005, 00006, 00007
994278.xbm	00000, 00001, 00002, 00003, 00004, 00005
996017.pptx	00201, 00266, 00894, 01413, 02775, 05439, 05498, 06316 06452, 06546, 06773, 06892, 07540, 07787, 08567

Table B.1: List of file fragments used for Experiment 3. For Experiment 2, the filenames are the same with the exception of the file extensions for the reasons discussed in the beginning of Chapter 4

THIS PAGE INTENTIONALLY LEFT BLANK

Referenced Authors

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudly Knox Library
   Naval Postgraduate School
   Monterey, California